16ビット・コンピュータをやさしく語る

はじめて読む 8086

村瀬康治 監修　蒲地輝尚 著



アスキー出版局

Photographed by Akio Kambayashi

はじめて読む 8086

村瀬康治 監修　蒲地輝尚 著

アスキー出版局

●本書を読む前に●

本書はMS-DOSの動作するパーソナルコンピュータを対象として，マシン語の学習を行います．本書のターゲットCPUは8086ですが，8088，80186，80286，V30(μPD70116-10)などのCPUについても本書の解説はほとんどそのままあてはまります．

MS-DOSは多くのパーソナルコンピュータに採用されており，しかもマシン語を勉強するのにとても適しています．本書でもMS-DOS上でマシン語を学習していきますが，「DIR」や「COPY」などの基本的なコマンドを使える程度の知識があれば本書を読み進めることができます．またそれらのコマンドを知らなくても，マシン語の学習そのものには差し支えありません．

本書ではマシン語を学ぶ道具(ツール)としてDEBUGコマンドを使用します．ただし，DEBUGコマンドがMS-DOSの標準システムに付属していない機種もありますので注意してください．DEBUGコマンドの代わりに，SYMDEBコマンドでも本書の実習はそのまま行うことができます．なお，MS-DOSのバージョンはどれを使ってもかまいませんが，8章の最後にあるサンプルプログラムに限りバージョン2.11以上を対象としています(2.11より古いバージョンはほとんど使われていない)．

実習のためのサンプルとして取り上げたシステムはNECのPC-9801シリーズ用のMS-DOSですが，ごく一部の実習を除いて，どの機種でも本書の実習をそのまま実行することができます．

監修のことば

本書は，拙著『はじめて読むマシン語』(8080，Z80を対象)を，蒲地輝尚君が8086CPUのためのマシン語入門書として書き直したものです．蒲地君は，拙著MS-DOS 3部作の『応用MS-DOS』の執筆の際に，全面的に協力していただいた青年ハッカーであり，彼を知る人は，彼のプログラミングのセンスが非凡であることをよく口にします．彼の見かけは「文学青年」(懐かしい言葉！)というイメージですが，わりといいオートバイに乗ったり，ビートルズのバンドでE.Guitarをやったり，けっこう凡人的Nowいこともしているようです．

さて，今までに8086のマシン語に挑戦した人の多くは，「そろそろマシン語の勉強を始めようと，8086についてのやさしそうな参考書を買ったけど，少し読み始めたところで行き詰まってしまった」，「Z80のマシン語は一応知っているが，8086になると何だか急にややこしくなって先に進めない」など，今一歩のところで立ち止まっているのではないかと思います．

確かに，8ビットCPUの8080やZ80のマシン語を知っている人にとっても，16ビットCPUの8086を理解するのは，そう容易ではないでしょう．ましてこれから始める人にとってはなおさらです．しかし本書を手にしたみなさんは，読み進むにつれ，マシン語やコンピュータの基本の1つ1つは，実に単純であることに気づくでしょう．そして本書に出逢った幸運をきっと感じていただけると信じています．

BASICの殻から一歩を踏み出し，マシン語――つまりはコンピュータの基本を学ぼうとしている賢明な読者に，心から励ましの言葉を送ります．

あなたと〝コンピュータ〟，そのほんとうの出逢いは本書から始まるのかも知れません．

1987年2月　　村瀬康治

はじめに

本書はマシン語の入門書です．しかし，けっしてマシン語だけの本ではありません．マシン語はコンピュータを直接コントロールできる唯一の言語であり，1つ1つの命令はコンピュータの仕組みと密接に結び付いています．ですからマシン語を学習するにはまず,コンピュータの仕組みやコンピュータ内部の各部の働きを知らなければなりません．この基礎知識があってこそマシン語の理解が可能なのです．

「16ビットマシンは8ビットマシンに比べて複雑でその仕組みは初心者には難解である」と一般には考えられているようです．このため，16ビットマシンのマシン語を初心者にも理解できるように，やさしく解説した本はあまりありませんでした．しかし16ビットマシンも，コンピュータとしての基本的な仕組みや，それを支える概念にまったく変わるところはありません．

本書は8ビットマシンのマシン語の経験の有無にかかわらず16ビットマシンのマシン語を理解することができるように，コンピュータの基礎的な知識からくわしく解説してあります．本書はマシン語命令のすべてを解説しているわけではなく，またマシン語のプログラミング技術の多くを教えているわけでもありません．しかし本書を読み終える頃には，今までブラックボックスであったコンピュータの中身がはっきりと見えてくるでしょう．コンピュータの基本的な仕組みを知ることこそ重要なことであり，本書の目的もそこにあります．本書をよく読んで，「ほんとうのコンピュータの姿」に触れてほしいのです．

MS-DOSやアセンブラなどの上級の解説書は，本書で述べている内容を基礎知識として要求しています．本書はそのような書物に挑戦するためにも十分役立つでしょう．

1987年2月　蒲地輝尚

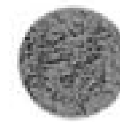
CONTENTS

Illustration：ツトム・イサジ

● 実習目次 ●

1

マシン語から広がる世界



本書の目的は 8086CPU のマシン語を解説することです．しかしマシン語を知らなくても，みなさんはコンピュータを，たとえばワープロに，またはデータベースにと，さまざまな目的で活用していることと思います．あるいは BASIC などの言語を使用して，自分でプログラムを作成しているかもしれません．では，なぜマシン語を学ぶ必要があるのでしょうか．まずそのことから話を始めなければなりません．

本書を手にしているみなさんは，コンピュータの仕組みに興味を持ち，コンピュータのことをもっと知りたいと思っているに違いありません．そのために必要となる知識がマシン語なのです．マシン語はコンピュータの仕組みとは切っても切れない関係にあります．

マシン語を学習することはコンピュータの仕組みを学習することであり，コンピュータの仕組みを学習することはマシン語を学習することでもあります．本章はそれを理解することが目標です．

1.1　コンピュータに直接命令できる唯一の言語 ―マシン語―

●

マシン語を学習する第一歩は，

コンピュータはマシン語でしか働かない

という事実を認識することから始まります．

みなさんは，すでに言葉の上ではこのことを知っているかもしれません．しかしコンピュータとマシン語の関係を的確に把握している人は少ないかと思います．言葉だけではなかなかピンとこないものです．

本節ではこのことを具体的な例を示しながら解説しますので，そのイメージをつかんでください．

MS-DOSを使っているみなさんは，「DIR」や「COPY」というコマンドを普段よく利用していることでしょう．これらのコマンドを入力すると，コンピュータはすぐさまファイルの一覧を表示したり，ファイルをコピーしたりします．表面的にはいかにもコンピュータがこれらのコマンドによって働くように思えます．

しかし，実はそうではありません．これらのコマンドは結果としてコンピュータを働かせますが，コンピュータそのものにはまったく通じないのです．コンピュータはマシン語しか理解できません．ですから「COPY　A：CONFIG.SYS　B：」のような人間の言葉に近い文章を入力しても働くわけはないのです．

では実際にどうしてマシン語ではないこれらのコマンドが機能するのでしょうか？　ここではそれを説明しましょう．

コンピュータが動作状態にあるときは，常になんらかのマシン語プログラムを実行しています．私たちは，そのマシン語プログラムに対して「DIR」な

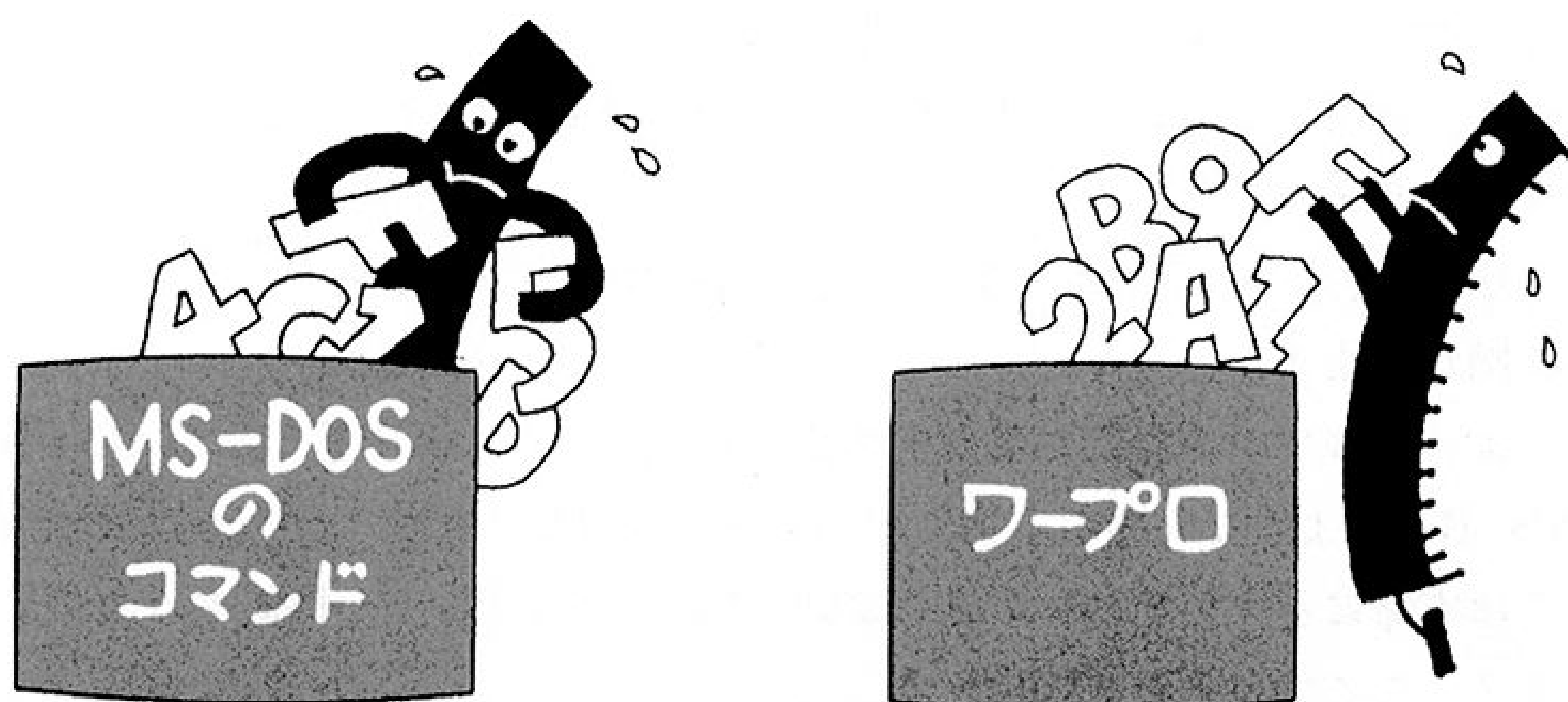
どのコマンドを与えているのです．私たちがコマンドを与えると，その結果，与えたコマンドを実行するためのマシン語プログラムに切り替えられて目的の動作が行われます．

コンピュータがコマンドの入力を待っている状態とは，コンピュータが「入力を待つマシン語プログラム」を実行している状態です．たとえば「DIR ⏎」と入力するとそのプログラムは「DIR」という文字列を解読して，それにより次に実行するプログラムを「ファイルの一覧を表示するマシン語プログラム」に切り替えます．そして，ファイルの一覧を表示するマシン語プログラムが実行された後は，再び「入力を待つマシン語プログラム」に切り替えられます．

この結果，あたかもコンピュータが「DIR」というコマンドで働いたかのように，ファイルの一覧が表示されるのです．

みなさんが使用しているワープロソフトやデータベースソフトなどもすべてマシン語プログラムです．ワープロを起動するコマンドを入力すると，「ディスクからマシン語プログラムをロードするマシン語プログラム」に切り替えられ，ワープロのプログラムがメモリに読み込まれます．そして実行するプログラムがそのワープロのマシン語プログラムに切り替えられるのです．

ワープロでは文書を入力し，ファイルに保存したり印刷したりすることができますが，コンピュータはみなさんの入力そのもので働いているのではな



＊インタープリタは同時通訳者という意味．マシン語よりも人間の言語に近いプログラミング言語を解釈しながら同時に実行するプログラムである．

く，あらかじめ用意されたマシン語プログラムをコマンドの入力によって切り替えて実行しているだけです．コンピュータは常に用意されたマシン語プログラムのうちのどれかを実行しているに過ぎません．

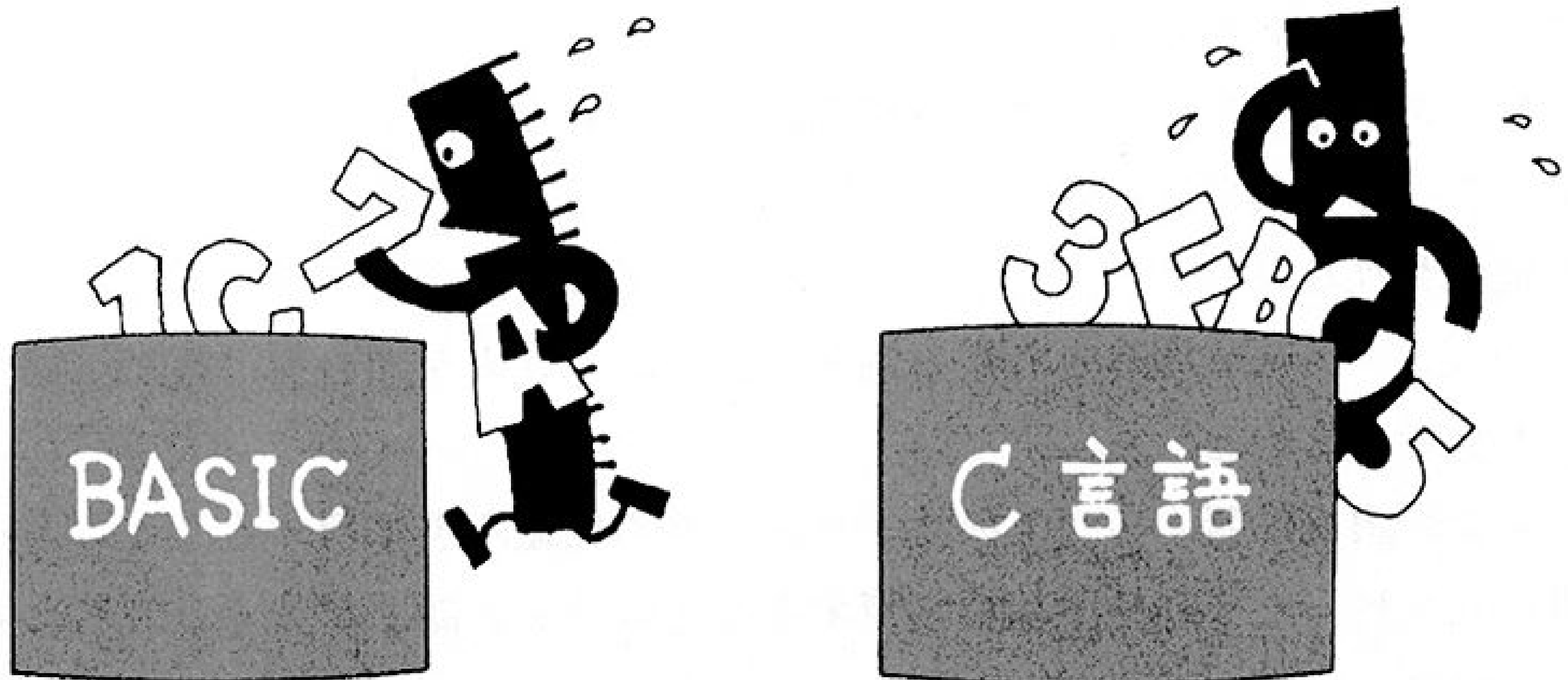
BASICなどのプログラミング言語についてもまったく同様です．BASICが起動した状態では，コンピュータはBASICインタープリタ*というマシン語プログラムを実行しています．

```
PRINT "ABC"
```

などというBASICのコマンドそのものではコンピュータは働きません．BASICインタープリタのマシン語プログラムがこの文字列を解読し，「文字列"ABC"を表示するというマシン語プログラム」へ切り替えているのです．

C言語**で書かれたプログラムについても同様です．コンピュータはC言語のプログラムそのものでは動きません．C言語で書かれたプログラムは，Cコンパイラ***というプログラムでマシン語プログラムに変換して初めてコンピュータを働かせることができるのです．

CコンパイラはC言語で書かれたプログラムを，マシン語プログラムに置き換えます．そうしてできあがったマシン語プログラムでなければコンピュータを働かせることはできないのです．



＊＊プログラミング言語の一種．
＊＊＊インタープリタが同時通訳者であるのに対し，コンパイラは翻訳家という意味を持つ．C言語で書かれたプログラムをまとめてマシン語プログラムに変換するプログラムである．

少しばかりややこしい話になりましたが，以上のことを図示してみましょう．

この図では中身が覗けますので，核となっているコンピュータの層が見えていますが，通常はこの表面しか見えないわけです．つまり私たちがコンピュータを使っているのは，この表面の「MS-DOSのコマンド」や「BASIC言語で書かれたユーザープログラム」をさわっているに過ぎません．

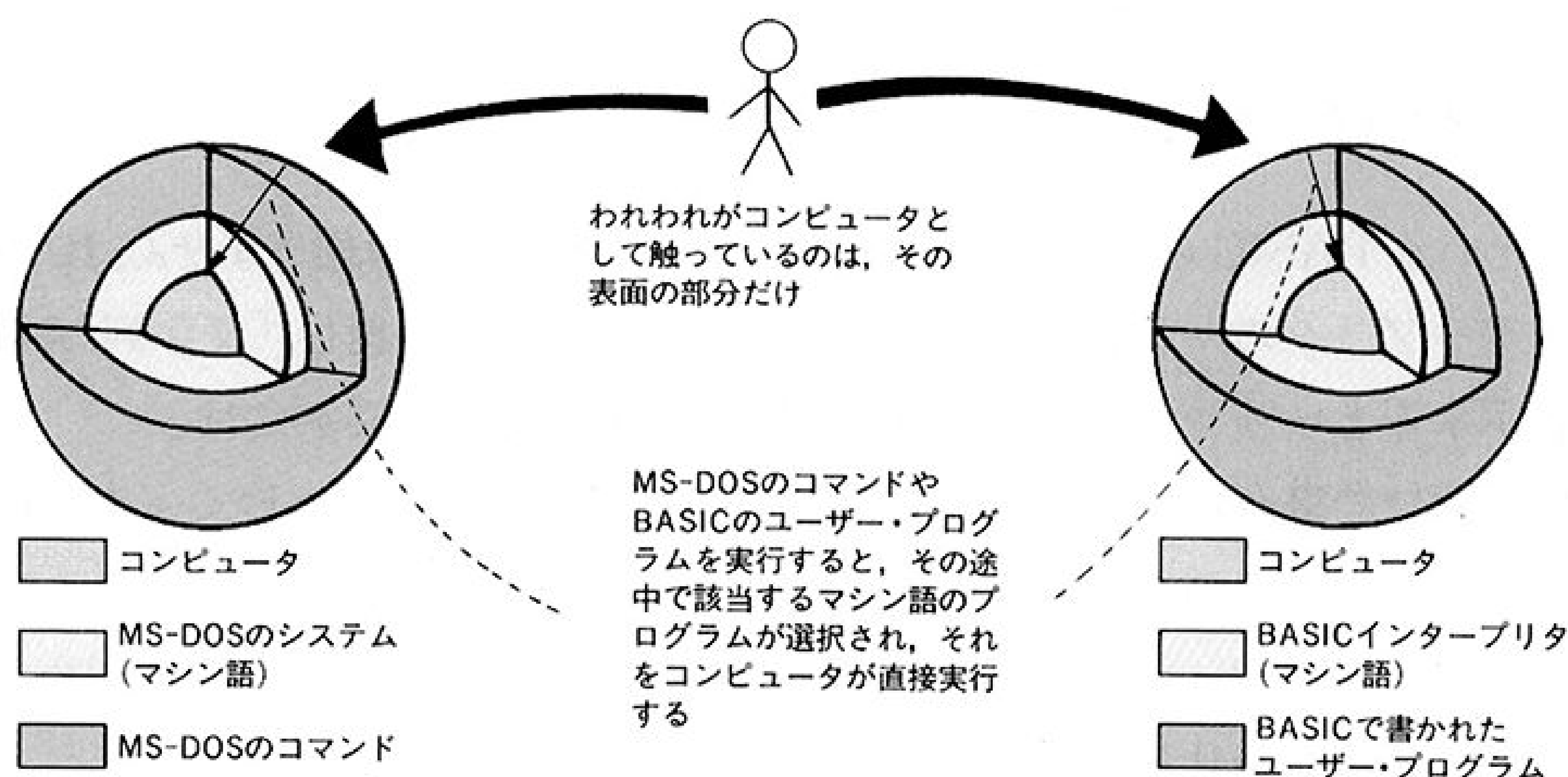


図 1-1　コンピュータとマシン語

マシン語はコンピュータを働かせる唯一の言葉であることが，わかってもらえたでしょうか．コンピュータに直接触れることができるのはマシン語だけです．どんなプログラムやコマンドであっても，最終的にはマシン語プログラムとして実行されているのです．

マシン語はコンピュータを直接働かせる言葉だからといっても，人間に理解できない言葉ではありません．人間の使う言葉に比べれば，むしろ非常に単純化された言葉です．ただしコンピュータに直接命令する言葉ですので，コンピュータの仕組みと密接な関係があり，それらを知らなければ理解することはできません．

マシン語を学習することはコンピュータの仕組みを学習することであり，逆にコンピュータの仕組みを学習することはマシン語を学習することにもなるのです．

マシン語のプログラムを学習することによって，コンピュータの仕組みを理解してください．そして，マシン語を通じてコンピュータの表面的な姿だけでなく，そのほんとうの姿に触れてほしいのです．

1.2　マシン語でコンピュータに命令しよう

私たちもマシン語を使うことによって，コンピュータに直接命令し，コンピュータを働かせることができます．それにはマシン語のプログラムを作成すればよいのです．それは，決してむずかしいことではありません．

みなさんが，これからマシン語を学習しようとするための準備は整っています．

MS-DOSはマシン語を学習するには実に恵まれた環境です．なぜならMS-DOSはもともとマシン語プログラム開発用のシステムだからです．もちろんそれだけを目的としたシステムではありませんが，マシン語プログラムを開発する環境として特に優れていると言えます．MS-DOSを使ってアプリケーションプログラムを使用しているみなさんにとっても，思いあたることがあるはずです．

MS-DOSのシステムディスクには，「FORMAT」や「DISKCOPY」といった基本的なコマンドのほかにもたくさんのコマンドがあります．なかにはまったく使ったことのないコマンドもいくつかあるでしょう．マニュアルを読んでも何のために使うコマンドなのかよく理解できなかったことと思います．たとえば次ページの図1-2のコマンドはどうでしょうか．

これらはすべてマシン語プログラムを開発するためのコマンドです．MS-DOSにはもともと，そのために必要な数々のコマンドが用意されているのです．

これらのコマンドは，プログラム開発に非常に強力な機能を発揮するだけでなく，マシン語を学習するためにも利用することができます．MS-DOSを使っているからには，これらのコマンドを活用しない手はありません．

ただし，一度にそのすべてをマスターする必要はありません．次章からはマシン語を学習するために，図1-2で色を付けて示した「DUMP（ダンプ）」コマンドと「DEBUG（デバッグ）」コマンドという2つのコマンドを使って実習を行います．みなさんのシステムディスクにもこの2つのコマンドが含まれていることを確認してください．

〈MS-DOS ver.3.1〉

```
A>DIR

 ドライブ A: のディスクにはボリュームラベルがありません
 ディレクトリは A:¥

COMMAND  COM     23942  85-12-11  15:28
ASSIGN   COM      1787  85-12-11  15:31
ATTRIB   EXE      8262  85-10-15  19:48
         COM             -10-21  18:52
DICM     COM     21248  85-
DISKCOPY COM      6880  85-10-14  14:00
DUMP     COM      1670  83-09-17  19:14
EDLIN    EXE      7394  85-12-11  15:37
EXE2BIN  EXE      2880  85-09-17   0:00
FC       EXE     14018  85-09-14  11:59
FIND     EXE      6483  85-10-14  17:32
FORMAT   EXE     34038  85-12-26  11:29
         EXE              -08-14  19:30
SORT     EXE      1680  85-
MSASSIGN COM      1509  85-07-11  15:59
MASM     EXE     77362  84-11-21  14:49
LINK     EXE     41114  84-11-14  14:48
SYMDEB   EXE     36538  86-01-08  17:54
MAPSYM   EXE     51904  85-06-21  10:21
CREF     EXE     10544  84-11-21  14:51
LIB      EXE     24138  84-10-31  16:57
MAKE     EXE     18675  84-08-13   1:24
KNJDIC   DRV      9797  85-10-24  23:20
NECDIC   DRV     31182  85-10-23  15:54
MOUSE    SYS      2989  85-05-30   3:24
MOUSE    DOC      2851  85-05-09  10:17
NECDIC   SYS    484352  85-10-24  13:48
KNJDIC   SYS     56320  83-09-20  14:22
CONFIG   SYS        20  85-10-25  13:39
README   DOC     16648  85-12-26  13:44
DEBUG    COM     11764  83-02-01  10:13
        47 個のファイルがあります
     34816 バイトが使用可能です
```

〈MS-DOS ver.2.1〉

```
A>DIR

 ドライブ A: のディスクにはボリュームラベルがありません
 ディレクトリは A:¥

COMMAND  COM     17190  83-09-19  16:19
ASSIGN   COM      1069  83-08-12  14:32
CHKDSK   COM      6774  83-09-04  23:29
COPY2    COM      3312  83-08-27  11:18
COPYA    COM      1319  83-09-13  19:46
CREF     EXE     13824  82-06-02  18:06
CUSTOM   COM      5476  85-06-21  15:03
DEBUG    COM     11764  83-02-01  10:13
DICM     COM     21248  85-05-21  12:54
DISKCOPY COM      6880  85-05-19  20:44
DUMP     COM      1670  83-09-17  19:14
EDLIN    COM      9124  83-09-17  12:57
EXE2BIN  EXE      1656  83-09-04  21:14
FC       EXE      2642  83-09-04  23:40
FIND     EXE      6430  83-09-04  23:41
FORMAT   COM     14742  85-05-19  13:59
KEY      COM      4531  83-10-30  19:46
LIB      EXE     23168  84-05-22  15:27
LINK     EXE     39952  84-04-13  16:03
MASM     EXE     81266  83-12-09  15:15
MORE     COM      4369  83-09-04  23:44
PRINT    COM      4697  83-09-18  18:02
RECOVER  COM      2607  83-09-04  23:51
                  1360  83-09-04  23:54
NECDIC   SYS
MOUSE    SYS      2989  85-05-30   3:24
MOUSE    DOC      2851  85-05-09  10:17
CONFIG   SYS        67  85-10-13  22:10
README   DOC      5334  85-06-27  11:42
        36 個のファイルがあります
    283648 バイトが使用可能です

A>
```

（網掛けの枠）はMS-DOSのマシン語プログラム開発用のコマンド

（白抜きの枠）は２章から７章のマシン語の実習で使用するコマンド．機種やMS-DOSのバージョンによっては，「DEBUG」の代わりに「SYMDEB」コマンドが含まれている場合もある．本書の実習は「SYMDEB」コマンドを使ってもまったく同様に行えるので，「SYMDEB」コマンドを持っている人は，そちらを使って実習する．

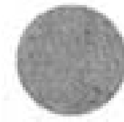
〈注意〉・この図は，PC-9801シリーズ用のMS-DOSを例とした．
・なお，機種によっては，これらのマシン語開発用のコマンドが別売になっている場合がある．

図 1-2　MS-DOS のマシン語プログラム開発用のコマンド

この２つのコマンドはマシン語プログラムを操作するためのたいへん基本的なコマンドであり，私たちはこれらのコマンドを使うことで簡単にマシン語を実習することができます．繰り返しますが，マシン語の学習は決してむずかしくありません．みなさんもこの２つのコマンドを駆使して，マシン語でコンピュータに命令してみましょう！

2

実行型ファイルをダンプする

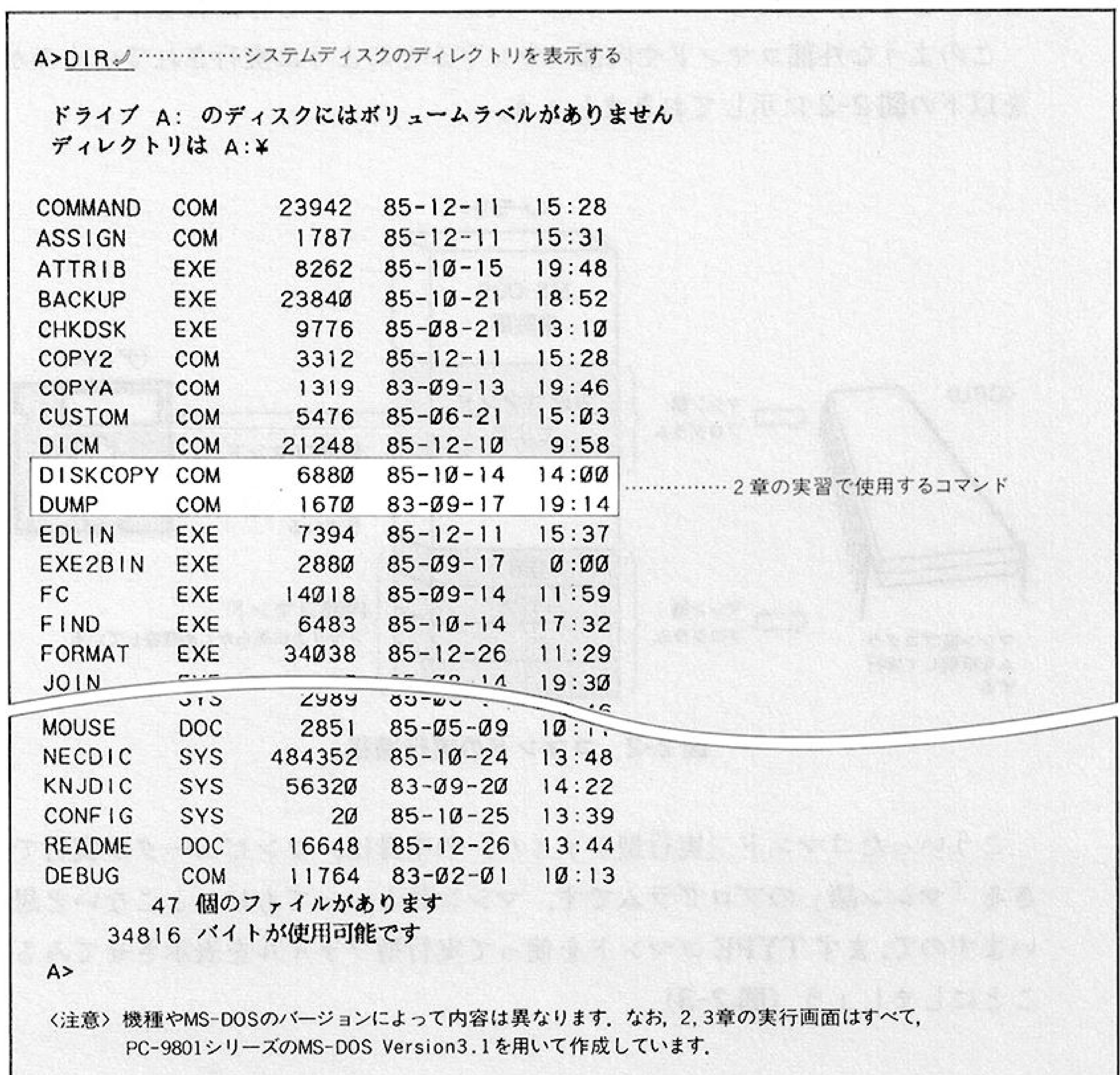


MS-DOSには，コンピュータが実行する「マシン語」のプログラムやデータを扱うためのコマンドがいくつか用意されています．コンピュータが実行するマシン語とは，人間にとってわかりやすい形式である「文字」ではなく，コンピュータにとってわかりやすい形式である「数値」で表されています．まず，この違いを理解することが，マシン語を理解することへの第一歩です．この章では，まだわからない用語などが出てくるかもしれませんが，それは後の章でくわしく解説します．とりあえず，ここではコンピュータ自身が扱う「マシン語」の姿を自分の目で実際に確かめてみることにしましょう．

この章ではMS-DOSのシステムディスク中のコマンドを使って実習を行います．そのためには必ずシステムディスクのバックアップをとり，そのバックアップディスクのほうを使ってください．そしてマスターディスクは安全な場所に保管しておきます．バックアップのとりかたは，マニュアル等を参照してください．

2.1 DISKCOPY.COM をダンプする —メッセージの確認

次の図 2-1 は, MS-DOS のシステムディスクのディレクトリを表示したものです.



```
A>DIR↵ ……………システムディスクのディレクトリを表示する

 ドライブ A: のディスクにはボリュームラベルがありません
 ディレクトリは A:¥

COMMAND  COM     23942  85-12-11  15:28
ASSIGN   COM      1787  85-12-11  15:31
ATTRIB   EXE      8262  85-10-15  19:48
BACKUP   EXE     23840  85-10-21  18:52
CHKDSK   EXE      9776  85-08-21  13:10
COPY2    COM      3312  85-12-11  15:28
COPYA    COM      1319  83-09-13  19:46
CUSTOM   COM      5476  85-06-21  15:03
DICM     COM     21248  85-12-10   9:58
DISKCOPY COM      6880  85-10-14  14:00 ……………2章の実習で使用するコマンド
DUMP     COM      1670  83-09-17  19:14
EDLIN    EXE      7394  85-12-11  15:37
EXE2BIN  EXE      2880  85-09-17   0:00
FC       EXE     14018  85-09-14  11:59
FIND     EXE      6483  85-10-14  17:32
FORMAT   EXE     34038  85-12-26  11:29
JOIN                                19:30
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
MOUSE    DOC      2851  85-05-09  10:
NECDIC   SYS    484352  85-10-24  13:48
KNJDIC   SYS     56320  83-09-20  14:22
CONFIG   SYS        20  85-10-25  13:39
README   DOC     16648  85-12-26  13:44
DEBUG    COM     11764  83-02-01  10:13
        47 個のファイルがあります
     34816 バイトが使用可能です
A>
```

〈注意〉 機種やMS-DOSのバージョンによって内容は異なります. なお, 2, 3章の実行画面はすべて, PC-9801シリーズのMS-DOS Version3.1を用いて作成しています.

図 2-1 MS-DOS のシステムディスクのディレクトリ

ディスクに入っているファイルの中で，「.EXE」や「.COM」の拡張子の付いたコマンドを「外部コマンド」と呼びます．これらのコマンドは，コマンドラインからコマンド名を入力することで，ディスクから指定した実行型ファイルがメモリ上に読み込まれ実行が開始されます．また，「マルチプラン」や「一太郎」などのアプリケーションソフトも，その実行方法はすべてMS-DOSの外部コマンドとまったく同等に扱われます．

これに対し，「DIR」や「TYPE」などのコマンドはMS-DOSのシステムとしてあらかじめメモリ上に置かれているので，ディスク上にはファイルが存在しません．これらのコマンドは「内部コマンド」と呼ばれます．

このような外部コマンドや内部コマンドがどのように実行されていくのかを以下の図2-2に示しておきましょう．

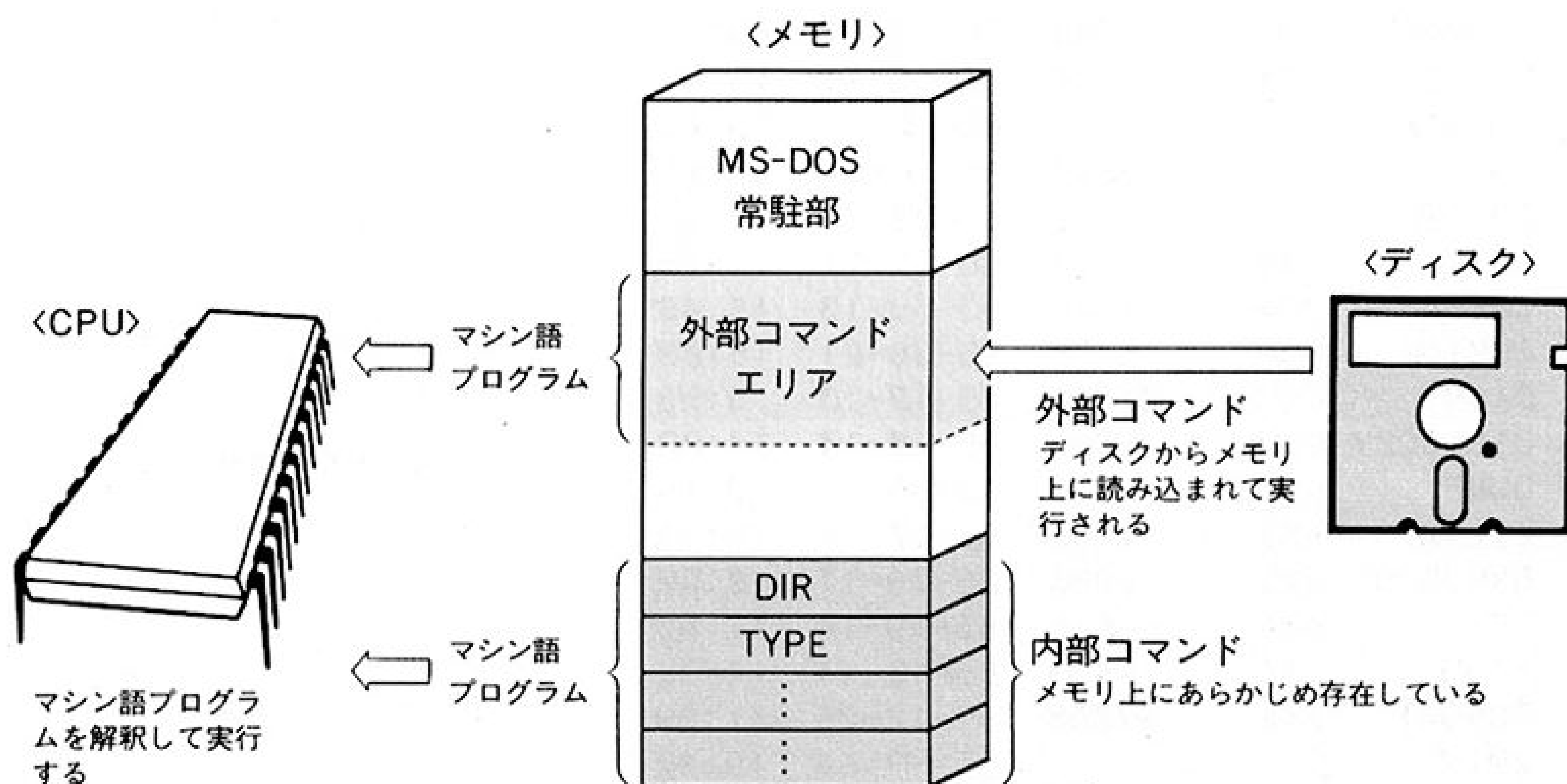


図2-2　コマンドの実行過程

こういったコマンド（実行型ファイル）の中身は，コンピュータが実行できる「マシン語」のプログラムです．マシン語といってもピンとこないと思いますので，まずTYPEコマンドを使って実行型ファイルを表示させてみることにしましょう（図2-3）．

日本語のメッセージが表示され，ビープ音が何度かして
このようにぐちゃぐちゃの画面になってしまう

```
_>m-t縦_>nVt尵_>o  t餡ｻ¥鑁｢uｻ|鏈｢ｦｻ煥ﾓ  Jﾍ!ｻ  ｴHﾍ!sｴHﾍ!｣ﾛ｣ﾝﾓ
  己ﾃ距+ﾐ｣ﾛ｣ﾝｺ <Zv  _>}  t
中止 <A> ,再試行 <R>s          ｾm濯彗  於6ｾ驍_>  tｾ
ｾｻ閥ｸﾍ!  ｾ汲関ｸﾍ!ｾ4鑵ﾆ"  s_宿_ｼﾞt鰮ｾｺ鰹ｾT鎰  ｶ tﾆ"｢簡_煣
uｾ汲 ｸﾍ!ｾ1阮  靜s
A>ラメータは無効です
ドライブの指定は無効です

記憶域が足りませんが挿入されていません

ディスクの種別が同じでなければなりません で読み込みを失敗しました で書き込みを
失敗しました の内容が異っています
```

A>……ここで「CLS」と入力すると画面がきれいに消去される
└─プロンプト

図 2-3　TYPE コマンドで DISKCOPY.COM を表示させると……

このように実行型ファイルは，テキストファイルのように TYPE コマンドを使って中身を表示させることはできません．TYPE コマンドというのは，私たちが読むことができる文字データでできているテキスト（文書ファイルやエディタで書いたプログラムファイルなど）を画面に表示するコマンドなのです．この TYPE コマンドに対して，MS-DOS には実行型ファイルの中身を覗いてみたり，編集したりする別のコマンドが用意されています．その1つに，DUMP（ダンプ）コマンドがあります．これは実行型ファイルの中身であるマシン語プログラムを人間の目で見える形で表示するコマンドです．言ってみれば，テキストファイルのかわりに実行型ファイルを表示する TYPE コマンドにあたります．

それでは実際に，このコマンドを使ってさきほどの「DISKCOPY.COM」ファイルの中身を表示させてみることにしましょう．DUMP コマンドは，次のように起動します．

A> DUMP ファイル名⏎

次の図 2-4 に実行結果を示します．なお，DUMP コマンドの表示は，TYPE コマンドと同様に [CTRL]+[S] によって一時中断および再開することができ，[CTRL]+[C] で途中終了させることができます．

```
A>DUMP DISKCOPY.COM↵ ……………「DISKCOPY.COM」ファイルを指定してDUMPコマンドを実行する
                        [CTRL]+[S] で表示の一時中断／再開，[CTRL]+[C] でプログラ
                        ムが途中終了し，MS-DOSに戻る
Dump Version 2.Ø


ØØØØØØØØ   E9 45 Ø4 ØØ ØØ ØØ ØØ ØØ-ØØ 2F Ø1 56 ØØ ØØ ØØ ØØ   .E......./.V....
ØØØØØØ1Ø   ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   ................
ØØØØØØ2Ø   ØØ ØØ ØØ ØØ ØD ØA 44 49-53 48 43 4F 5Ø 59 2Ø 76   ......DISKCOPY v
ØØØØØØ3Ø   65 72 73 69 6F 6E 2Ø 32-2E 31 ØØ ØD ØA 82 63 82   ersion 2.1....c.
ØØØØØØ4Ø   ...                  5B 83 57 83 87 83 93 82   n.r.7.o.[.W.....
                                          97 82 E8   ｪ...｢.ﾜ.ｷ

数字や文字の組み合せが同じように繰り返し表示される

...Ø1A6Ø   12 B4 2C CD 21 89 ØE 8E-12 89 ...
ØØØØ1A7Ø   8A DØ 8Ø E2 1F B1 Ø5 D3-E8 8A FØ 8Ø E6 ØF A1 12   .ﾐ...ﾜ.ﾓ........
ØØØØ1A8Ø   12 B1 Ø9 D3 E8 8B C8 81-C1 BC Ø7 B4 2B CD 21 A1   .ﾜ.ﾓ..ﾈ.ﾁｼ.ｴ+ﾍ!.
ØØØØ1A9Ø   1Ø 12 32 D2 8A FØ 8Ø E6-1F DØ E6 B1 ØB D3 E8 8A   ..2ﾒ.....ﾐ.ﾜ.ﾓ..
ØØØØ1AAØ   E8 A1 1Ø 12 B1 Ø5 D3 E8-8A C8 8Ø E1 3F B4 2D CD   ....ﾜ.ﾓ..ﾈ..?ｴ-ﾍ
ØØØØ1ABØ   21 C6 Ø6 9Ø 12 Ø1 58 5A-59 FB C3 FA 8B ØE 8A 12   !ﾆ....XZY.ﾃ.....
ØØØØ1ACØ   8B 16 88 12 B4 2B CD 21-8B ØE 8E 12 8B 16 8C 12   ....ｴ+ﾍ!........
ØØØØ1ADØ   B4 2D CD 21 C6 Ø6 9Ø 12-ØØ FB C3 ØØ ØØ ØØ ØØ ØØ   ｴ-ﾍ!ﾆ.....ﾃ.....

A>
        〈注意〉機種やMS-DOSのバージョンによって内容は異なります
```

図 2-4　DISKCOPY.COM をダンプする

TYPE コマンドではきちんと表示できなかった「DISKCOPY.COM」ファイルの中身が次々と表示されたことがわかるでしょう．DUMP コマンドを使ってファイルの中身を表示することを「ダンプをとる」と言います．また，このような表示のことを「ダンプリスト」と呼びます．

さて，このダンプリストがいったい何を意味しているのかを説明していきましょう．**図 2-4** を見ればわかるように，画面には数字，英文字，ピリオド(.) などがたくさん表示されています．また，ダンプリストは 1 行ごとに同じような形式で繰り返されているのがわかると思います．各行は左から，8 文字の数字があり，2 文字ずつの数字やアルファベットの組が 8 組ごとにハイフンで区切られて 16 個並び，そして右端にはピリオド(.)が随所に混じっているさまざまな文字が 16 個並んでいます．

このなかで，まん中に並んでいる 2 文字ずつの数字とアルファベットの組がマシン語のプログラムです．コンピュータはこのような数字とアルファベットの組で表されるプログラムを解釈して実行するのですが，私たちはこれだけ見ても何のことだかさっぱりわかりません．実は，この 1 つの組は 16 進法で表した数値，すなわち **16 進数**なのです．16 進数については 4 章でくわし

く説明しますのでここではわからなくてもけっこうです．とにかく，この2文字の組合せはコンピュータが実行可能なマシン語のプログラムであるということを理解してください．

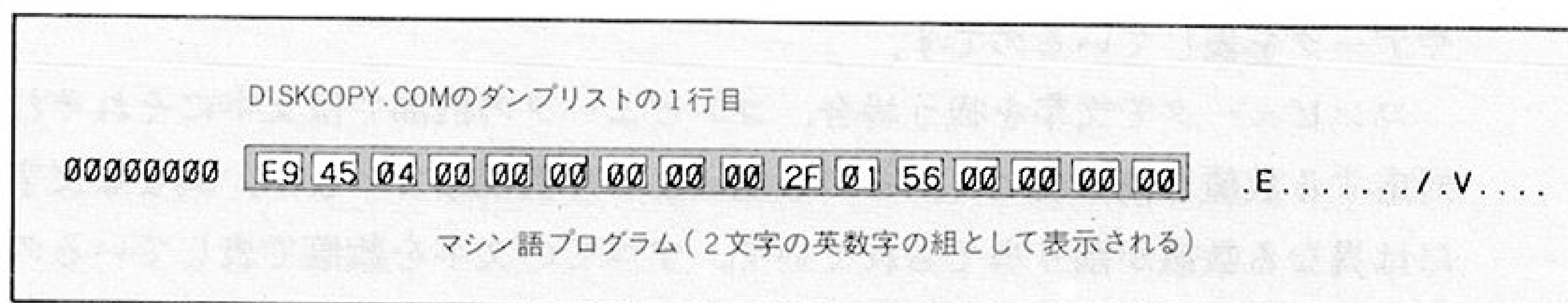
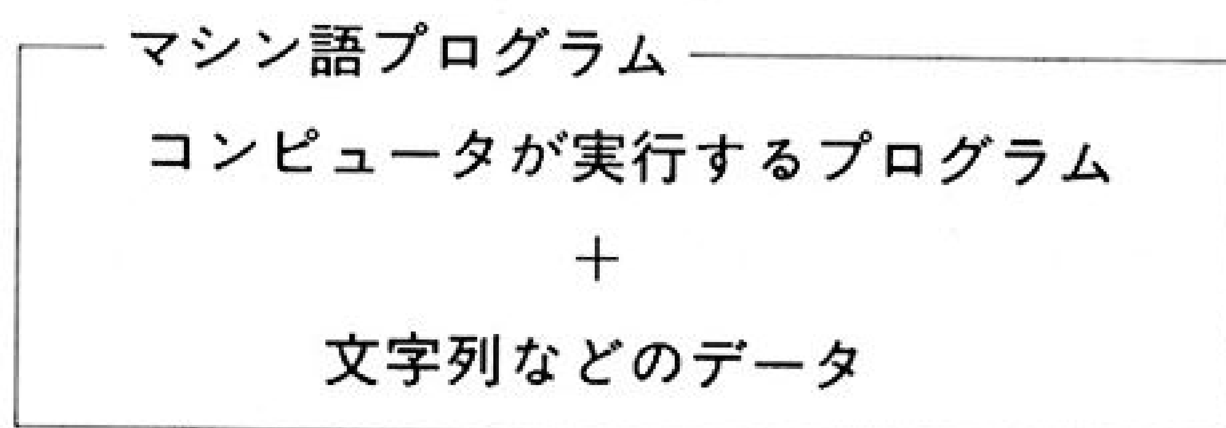


図2-5　ダンプリスト中のマシン語のプログラム

16進数については，私たちが普段使っている10進数とは違った数字の読み方をします．このダンプリストの先頭にあるデータは「E9」ですが，これは「イーきゅう」と読んでください．その次のデータは「45」ですが，「よんじゅうご」ではなく「よんご」と1文字ずつ読みます．

このように16進の数字2文字で表せる数の単位を**1バイト**と呼びます．1バイトのデータが並んだものがマシン語プログラムの姿なのです．ここでは一言でマシン語プログラムと呼んでいますが，実際には1バイトのデータの並びのうち，ある部分はコンピュータが実行するマシン語のプログラムそのものですし，またある部分はそのプログラムで使われる文字列などのデータです．



しかし，先の図2-4を見る限りその区別はつかず，どちらも2文字の16進数でしかありません．

なお，以下で「データ」という言葉を使う場合には，この両者をひっくるめたデータ全般という意味と，プログラムで使用する文字列などのデータという意味の2つの場合がありますから，注意してください．

ダンプリストの右端には，ピリオドのたくさん混じった文字が並んでいますが，これは左側にある1バイトの数値にそれぞれ対応しています．2文字からなる1バイトの数値の数と，ピリオドを含む文字の数はともに16個と同じであることに気づくでしょう．両者はまったく同じマシン語のプログラムやデータを表しているのです．

コンピュータで文字を扱う場合，コンピュータの内部では文字にそれぞれ対応する数値を割り当てて，その数値によって表現しています．異なる文字には異なる数値が割り当てられていて，すべての文字を数値で表しているのです．それぞれの文字に対応する数値を「キャラクタ（文字）コード」と呼びます． 各文字と16進の数値との対応表を273ページのAPPENDIXに載せておきますので参照してください．

ダンプリストの右端に表示されているのは，数値で表されるマシン語のプログラムやデータを，キャラクタコードと考えた時のその数値に対応する文字です．両者は同じ順番に並んだ同じデータを表しており，1つ1つが1バイトのデータに対応しています．1バイトで表せる数値のなかには，画面に表示できない文字もあり，そういう数値に対しては，ピリオド（.）が表示されています．

ただし，誤解しないように注意することは，必ずしも右側にある文字そのものがプログラム中に文字として含まれるわけではありません．たとえば，次の図2-6にあるように2組目の「45」という数値は右側を見ると「E」の文字に対応していますが，これは実際に「E」という文字データではなくマシン語プログラムの一部です．

これらはマシン語プログラムの一部であって文字列データではない

```
00000000  E9 45 04 00 00 00 00 00-00 2F 01 56 00 00 00 00   .E......./.V....
00000010  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000020  00 00 00 00 0D 0A 44 49-53 4B 43 4F 50 59 20 76   ......DISKCOPY v
```

D I S K C O P Y ␣ v

これらは文字列である

キャラクタコード表に対応する文字が画面に表示できない文字のときはピリオドが表示される

図2-6　マシン語プログラムと対応する文字

この場合は，プログラムの一部がたまたま「E」のキャラクタコードと同じであっただけです．それでは，実際にマシン語プログラムで使われる文字データを確認してみましょう．図 2-7 のダンプリストで示すように先頭の 3 行目から 4 行目にかけて「DISKCOPY version 2.1」という文字列があることがわかります．みなさんのシステムではどうでしょうか？　これは DISKCOPY コマンドを起動したときに表示されるメッセージの一部そのものです．

```
A>DUMP DISKCOPY.COM↵ ..............もう一度，DISKCOPY.COMをダンプしてみる

Dump Version 2.0

00000000  E9 45 04 00 00 00 00 00-00 2F 01 56 00 00 00 00   .E......./.V....
00000010  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000020  00 00 00 00 0D 0A 44 49-53 4B 43 4F 50 59 20 76   .....DISKCOPY v
00000030  65 72 73 69 6F 6E 20 32-2E 31 00 0D 0A 82 63 82   ersion 2.1....c.
00000040  ^C .............. [CTRL]+[C] で途中終了する

A>DISKCOPY A: B:↵ ..............DISKCOPYコマンドを実行してみる

DISKCOPY version 2.1 ← マシン語プログラムに含まれていた文字列データが表示されている

ディスクのコピーを行います

送り側ディスクをドライブ A: に挿入してください
受け側ディスクをドライブ B: に挿入してください
準備ができたらどれかのキーを押してください^C ............表示されるメッセージを確認したら，
                                                            [CTRL]+[C] で途中終了する
A>
```

図 2-7　DISKCOPY.COM の起動時のメッセージ

このように，DUMP コマンドを使うことによって，実行型ファイルの中にある文字列データを調べることができます．

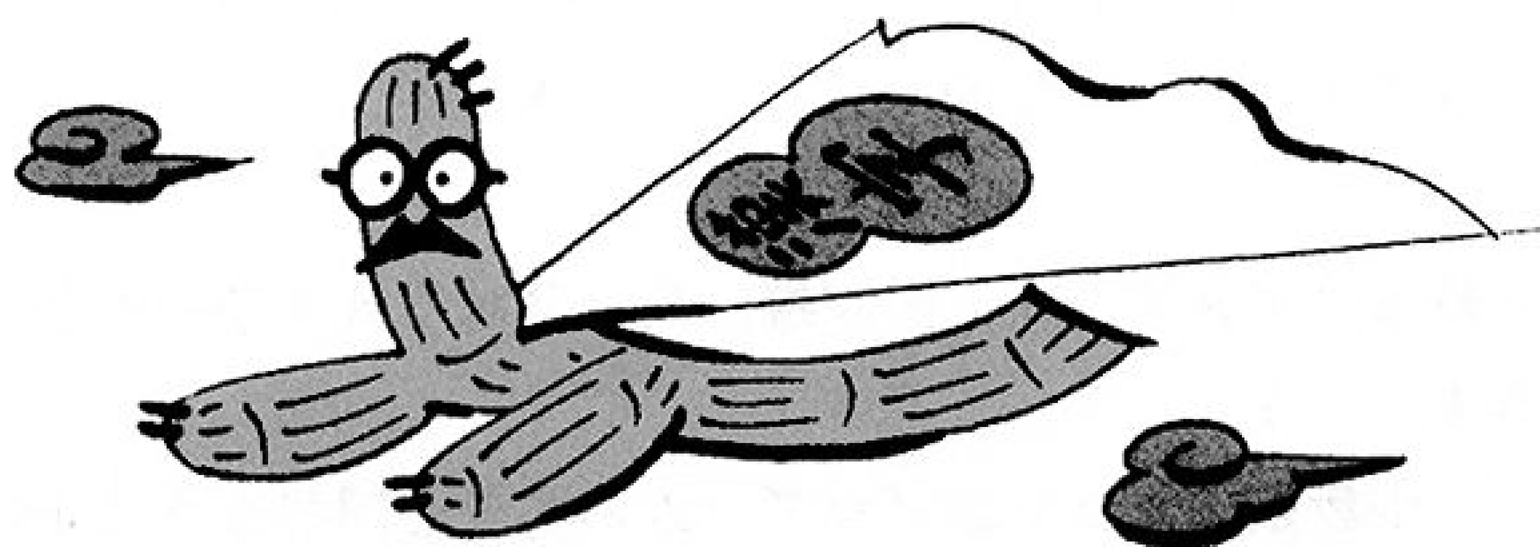
最後にダンプリストの左端に表示される 8 文字の数字について説明しておきましょう．

この数値はマシン語プログラムの場合と同じように 16 進の数値で，ファイルの先頭からの位置を表しています．つまり，その行の先頭のデータが，ファイルの先頭から何バイト目（何番目）のデータであるかを意味しているのです．

2.2 DISKCOPY.COM の漢字のメッセージを確認する

前節で DISKCOPY.COM のバージョンナンバーを示すメッセージを確認することができました．DISKCOPY コマンドを実行すると，「送り側ディスクをドライブ A：に挿入してください」などの漢字のメッセージも表示されます*．

バージョンナンバーを示すメッセージがアルファベットと数字のみであったのに対し，このメッセージは漢字を含むメッセージです．もちろん，漢字のメッセージもマシン語プログラム中のデータであるわけですが，先の DUMP コマンドでは表示されませんでした．そこで，本書では実行型ファイルの中の漢字のメッセージを確認するために，CDUMP（シーダンプ）というコマンドを使用します．このコマンドは本書の 8 章で作成するものですが，ここではとりあえず，以下で示す画面を見てもらえば結構です．DUMP コマンドで漢字も表示されるバージョンをお持ちの方は，実際に DUMP コマンドを使って同じように操作をしてみてください．また，MS-DOS のツールとして CDUMP に相当するコマンドを持っている方もいるでしょう．



＊機種や MS-DOS のシステムによって，このような漢字のメッセージが表示されないものもある．

それでは，CDUMP コマンドで DISKCOPY.COM をダンプしてみましょう．CDUMP コマンドの使用方法は，DUMP コマンドと同じように起動時にファイル名を指定します．実行結果を図 2-8 に示します．

```
A>CDUMP DISKCOPY.COM↲ ……「DISKCOPY.COM」ファイルを指定してCDUMPコマンドを実行する
                        [CTRL]+[C] で表示の中断/再開，[CTRL]+[S] でプログラムを途
                        中終了し，MS-DOSに戻る
000000: 颱......./.V........................DISKCOPY version 2.1...DO
000041:  Sのバージョンが違います...送り側ディスクをドライブ @: に挿入し
000080: てください...受け側ディスクをドライブ @: に挿入してください..準備
0000C1:  ができたらどれかのキーを押してください .....ディスクのコピーを行
000101:  います.......ディスクの照合を行います......コピーは終了しました
000141:  .....照合を終了しました...別のディスクをコピーしますか <Y/N>? .
000180: ..別のディスクを照合しますか <Y/N>? .....コピーは失敗しました...
0001C0: ..照合は失敗しました.....送り側ディスク.....受け側ディスク...中止
000201:   <A> ,再試行 <R> ,強行 <I> ? ...中止 <A> ,強行 <I> ? .....パラメ
000241:  ータは無効です.....ドライブの指定は無効です.....記憶域が足りませ
000281:  ん.が挿入されていません........ディスクの種別が同じでなければな
0002C1: りません.で読み込みを失敗しました.で書き込みを失敗しました.の
000301:  内容が異っています.....トラック .は不正です...................
000340: ..............................................................
000380: ..............................................................
0003C0: ..............................................................
000400: ..............................................................
000440: ........弾.厖紗ｼF.S抉ｾ$.鎊.ｴ.ﾍ!｢...ｸ@.実&..$.｢..ｾｬ.&ｬ｢..&ｬ｢..&ｬ
000480: ｢..&ｬ｢..&..$.u.ｾ:.  精.7ﾍ!.......峪...ﾃ..t.ﾆ.....X...........
0004C0: ......u.ﾆ.¥..ﾆ.|....ｦ...<.t.......t.ｾP.雜.絣.ｾ..齶.s..=逗<:t...灰
000500: .$°<As.  <Zv.  ｬ< t.<.u...覩..<:u.灰.$°<As.  <Zv.  .>].  t..>].-t
000540: .鶻..>^.Vt.驟..>_. t.鰥..>m. t..>m.-t.鰲..>n.Vt.旭..>o. t.餾..ｻ¥.
000580: 錚｢ .. ｲ!  錚 ｢....ｦ.ｻ..ｱ.ﾓ  Jﾍ!ｻ..ｴHﾍ!s.ｴHﾍ!｣ﾛ....｣ﾝ.欣..己.ﾃ
        ........6.ｾ...>...t.ｾ..騒.....u.ﾆ
```

16進数のマシン語プログラムは表示されずに，文字だけが横に広く表示される

```
0017C0: .;...t.薫.=..t....ﾝ...ﾝ.ｾ..ｿ.........................!s.
001800: 鵑.鏞.ｺr.ｴ.ﾍ!ﾆ....<.u.ﾆ....鎚.ｺ..ｴ.ﾍ!<.t..>...u.鬮.鰈..>...t.驪.
001840: .>...u..%声..ｴ:ﾍ!s.髯.ｺ..ｴ:ﾍ!s.彫.....酉.嶌.  ｾ)..>...t.ｾB..|...
001880: ｾ..>...t.ｾ........t....v.-#ｴ%ﾍ!..>x..t.ｺ..ｴ:ﾍ!.>...t..>y..t.ｺ&
0018C0: .ｴ;ﾍ!.>...t........ｴ+........ｴ-ﾍ!.....ｴ.ﾍ!ｴ.ﾍ!ｾ*.隰.ｾM.腥.
001900: 2ﾒｴ2ﾍ!季..｣n....n.瀬外.2  W....ﾛ.3ﾛ^%DD...ﾛ.ｻ.....n..ﾇ.p....?.u.
001940: 絵.$.<.u....p.|t.ε..?.u.絵.$.<pu...p.C|t.  ﾃｺ..ｴ.ﾍ!ﾃｺ....  ..ｾ.
001980: .ｿﾌ.ｹ...ｬ< t.ｪ欄ｾ..ｹ..ｬ< t.瘋ｰ..ｪ監ｪ|ｬ< t.ｪ欄ﾆ..ﾃｾ"...ｾ....ｿﾛ.ｹ..
0019C0: .ｬ< t.ｪ欄..|.ｹ..ｬ< t.瘋ｰ.ｪ|監ｪｬ< t.ｪ欄ﾆ.ﾃ...ﾛ.ｾ...ｾr.......ﾛ.
001A00: ｾ.....ｾ.......ｹ(...ﾃｹ,..ｸT.ﾃ...ｿ.....ｿr......ｿ.....ｿ......ﾛ.
001A40: T寓ｹ(...ﾃｹ,...ﾃ.QRPｴ※ﾍ!........ｴ,ﾍ!.........諌.....7.ﾓ闕.....
001A80: .7.ﾓ閑ﾈ  ｼ.ｴ+ﾍ!...2ﾒ婿...ﾐ豎.ﾓ閧旁..7.ﾓ閧ﾈ..?ｴ-ﾍ!ﾆ.....XZY.ﾃ.....
001AC0: ....ｴ+ﾍ!.........ｴ-ﾍ!ﾆ.....ﾃ.....

A>
```

〈注意〉第２水準漢字ROMを搭載していない機種では，一部の表示が異なります

図 2-8　DISKCOPY.COM を CDUMP する

漢字のメッセージが表示されているのがわかりますね．ただし，CDUMP コマンドでは，DUMP コマンドのときに表示された 16 進数のマシン語プログラムが表示されず，そのデータに対応する文字だけが横に広く表示されています*．また行の左端の数値は DUMP コマンドと同様に，その行の最初の文字がファイルの先頭から何バイト目（何番目）であるかを示しています．これを DUMP コマンドのリストと比較すれば，どの文字がどういう数値に対応するのかを調べることができるのです．

それでは，DUMP コマンドと CDUMP コマンドによるダンプリストの対応を次の図 2-9 に示してみましょう．

〈CDUMPコマンドのダンプリスト〉　CDUMPの各行はDUMPコマンドでは下の範囲を表す

```
000000:  螂......../.V............................DISKCOPY version 2.1...D O
000041:   S の バ ー ジ ョ ン が 違 い ま す ...送 り 側 デ ィ ス ク を ド ラ イ ブ  @: に 挿 入 し
000080:  て く だ さ い ...受 け 側 デ ィ ス ク を ド ラ イ ブ  @: に 挿 入 し て く だ さ い ..準 備
0000C1:   が で き た ら ど れ か の キ ー を 押 し て く だ さ い  .....デ ィ ス ク の コ ピ ー を 行
```

〈DUMPコマンドのダンプリスト〉

```
00000000  E9 45 04 00 00 00 00 00-00 2F 01 56 00 00 00 00   .E......./.V....
00000010  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000020  00 00 00 00 0D 0A 44 49-53 4B 43 4F 50 59 20 76   ......DISKCOPY v
00000030  65 72 73 69 6F 6E 20 32-2E 31 00 0D 0A 82 63 82   ersion 2.1....c.
00000040  6E 82 72 82 CC 83 6F 81-5B 83 57 83 87 83 93 82   n.r.7.o.[.W.....
00000050  AA 88 E1 82 A2 82 DC 82-B7 00 0D 0A 91 97 82 E8   ｪ...｢.ﾜ.ｷ.......
00000060  91 A4 83 66 83 42 83 58-83 4E 82 F0 83 68 83 89   .､.f.B.X.N...h..
00000070  83 43 83 75 20 40 3A 20-82 C9 91 7D 93 FC 82 B5   .C.u @: ./.}...ｵ
```

図 2-9　DUMP と CDUMP のダンプリストの対応

さて，CDUMP コマンドのダンプリスト中の漢字データは，どのような数値に対応するのでしょうか？　アルファベットによるバージョンナンバーのメッセージは CDUMP コマンドのダンプリストでも確認することができます．それと DUMP コマンドのダンプリストを見比べて見てください．

まず CDUMP のダンプリストを見てみると，「……version 2.1」という文字列に続いて 3 つのピリオド（.）があり，そのあとに「ＤＯＳのバージョンが違います」というメッセージがあります（図 2-10）．

---

*漢字バージョンの DUMP コマンドを実行している場合は，16 進数のマシン語プログラムも同時に表示される．

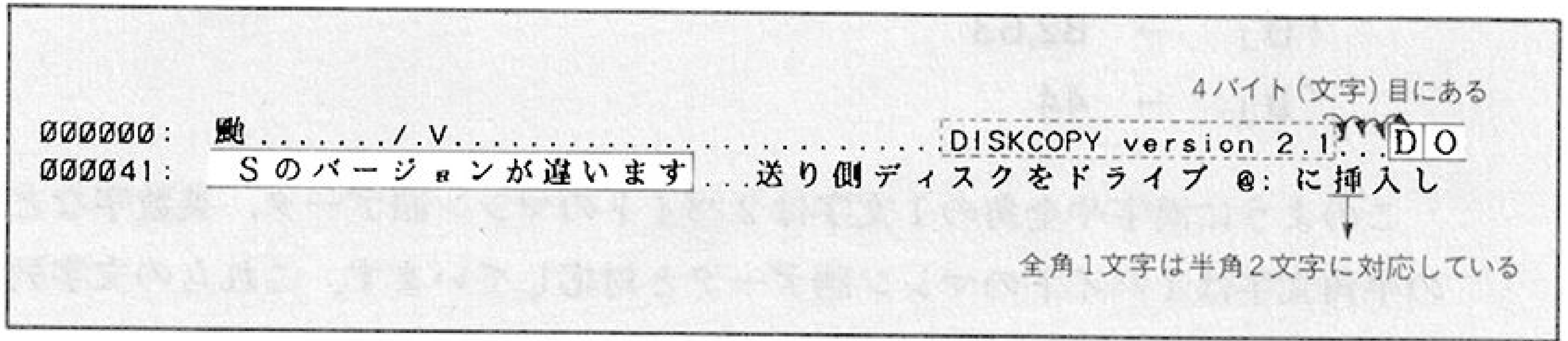


図2-10　CDUMPコマンドによるメッセージの確認

この3つのピリオドは，3バイトのデータがあることを意味しています．それでは今度はDUMPコマンドのダンプリストを見てみましょう(図2-11)．

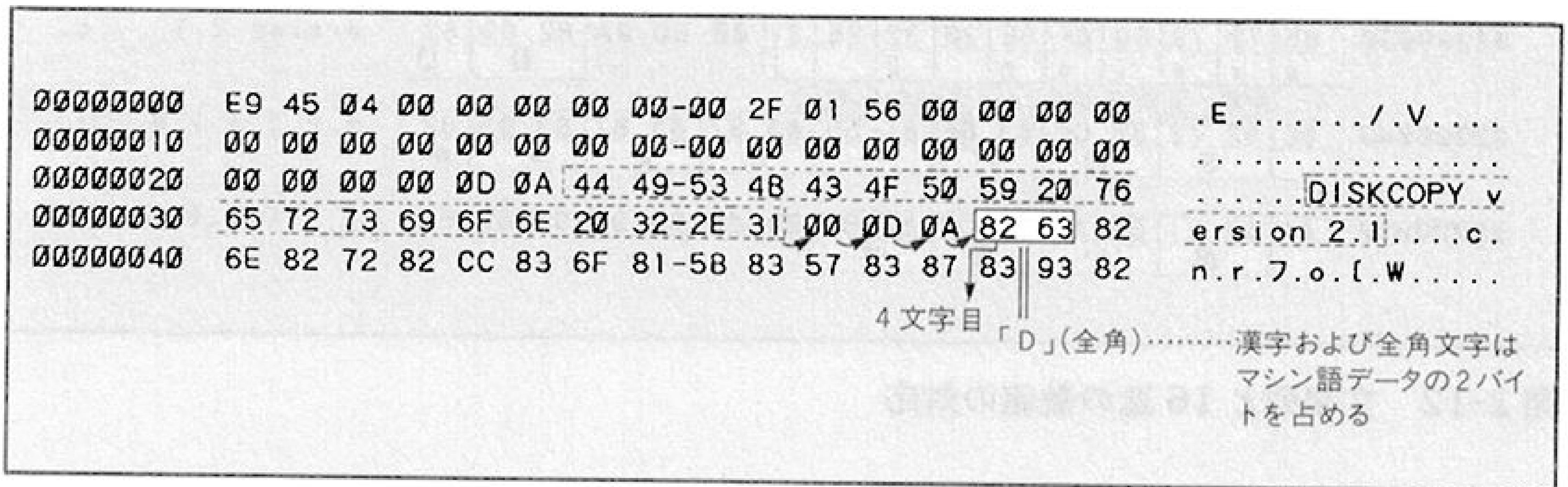


図2-11　DUMPコマンドによるメッセージの確認

図2-10でわかるように，「DOSのバージョンが……」の「D」(全角)*の文字は「……version 2.1」から4バイト目にありました．これに対応する図2-11のダンプリストのデータは，4行目の右から3番目の数値「82」となっています．ここでCDUMPのダンプリストをよく見ると，漢字や全角文字の1文字は2バイト分のデータを占めていることがわかります．ですから，「D」は，「82」とそれに続く「63」という2つの16進数に対応していることになります．つまり，「D」は「82，63」という2バイトの数値として表されているのです．

これに対し，「DISKCOPY version 2.1」の先頭にある半角*の「D」は，図2-11を見ると，「44」という数値に対応しています．

*半角文字とはコマンドを入力するためなどに使用する英文字や数字のことで，全角文字とは漢字や漢字と同じ大きさの英数字，ひらがななどのことである．

「Ｄ」 → 82,63
「D」 → 44

このように漢字や全角の１文字は２バイトのマシン語データ，英数字などの半角文字は１バイトのマシン語データと対応しています．これらの文字列と 16 進の数値との対応関係を図 2-12 に示してみました．


```
00000020  00 00 00 00 0D 0A 44 49 53 4B 43 4F 50 59 20 76   ......DISKCOPY v
                            D  I  S  K  C  O  P  Y     v
00000030  65 72 73 69 6F 6E 20 32 2E 31 00 0D 0A 82 63 82   ersion 2.1....c.
          e  r  s  i  o  n     2     1          Ｄ    Ｏ
00000040  6E 82 72 82 CC 83 6F 81 5B 83 57 83 87 83 93 82   n.r.7.o.[.W.....
             Ｓ    の    バ    ー    ジ    ョ    ン    が
00000050  AA 88 E1 82 A2 82 DC 82 B7 00 0D 0A 91 97 82 E8   ｪ...｢.ﾜ.ｷ.......
             違    い    ま    す
```

図 2-12　文字列と 16 進の数値の対応

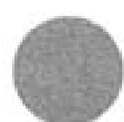
どの漢字にもこのように数値が割り当てられています．これも一種のキャラクタコードで，MS-DOS で使われている漢字コードは「シフト JIS コード」と呼ばれるものです．漢字の各文字がどんなコードに対応しているかは，MS-DOS のマニュアル等に一覧表が出ています．

ただし，このような文字のキャラクタコードや漢字のシフト JIS コードを覚えたり，コード表を索くことは，実際にはほとんど必要ありません．文字や漢字を入力するとコンピュータの内部では自動的に対応するコードに変換され，表示されるときにはそのコードが英数字や漢字となるからです．

この章では，マシン語のプログラムが 16 進数で表されているということと，マシン語プログラム中に含まれる文字列データは DUMP コマンドや CDUMP コマンドを使って表示させることができるということを頭に入れておいてください．

3

実行型ファイルのメッセージを変更する



前章でDUMPおよびCDUMPコマンドの使い方とその役割を説明しました．この章ではマシン語プログラムを扱うためのさらに強力なコマンドである，DEBUGコマンドについて解説します．

DEBUGコマンドでは，DUMPコマンドのようにマシン語プログラムを16進数で表示することはもちろん，マシン語プログラムを入力／変更したり，それを実行したりなど，いろいろなことができます．DEBUGコマンドは，「マシン語」を扱うためにはなくてはならないコマンドであり，逆に，DEBUGコマンドが使いこなせるようになれば，コンピュータの仕組みがかなりわかってきたと言ってよいでしょう．

この章ではDEBUGコマンドの基本的な使い方を解説し，実際にマシン語プログラム中のデータを操作する例として，DISKCOPYコマンドのメッセージを一部変更してみます．

ここで1つ，〝注意！〟があります．この章ではメモリの内容を直接書き換えたり，ディスクへの書き込みを行ったりしますので，この実験に使うディスクは必ずバックアップをとったディスクを使ってください．実験でファイルの内容を書き換えてしまいますし，あるいはキーの入力ミスなどでディスクの内容を破壊するかもしれないからです．ここでは，手順を追いながら解説していきますから，みなさんも1つ1つ確認をしながら作業を進めてください．

3.1　DEBUGコマンドの機能と使い方

テキストファイルには，その内容を編集するためのコマンドとして「EDLIN」などのエディタがあります．これに対して，ここで紹介する「DEBUG」* コマンドは，メモリの内容を操作するためのエディタであると思ってもらえばよいでしょう．

2章で外部コマンド（実行型ファイル）はメモリ上に読み込まれてから実行されるということを述べましたが，このメモリ上にあるマシン語プログラムを操作／編集するためのコマンドが「DEBUG」なのです．メモリ上の値を扱うという点が，ファイルを読み込んで表示するだけのDUMPコマンドやCDUMPコマンドとは大きく異なっています（図3-1）．

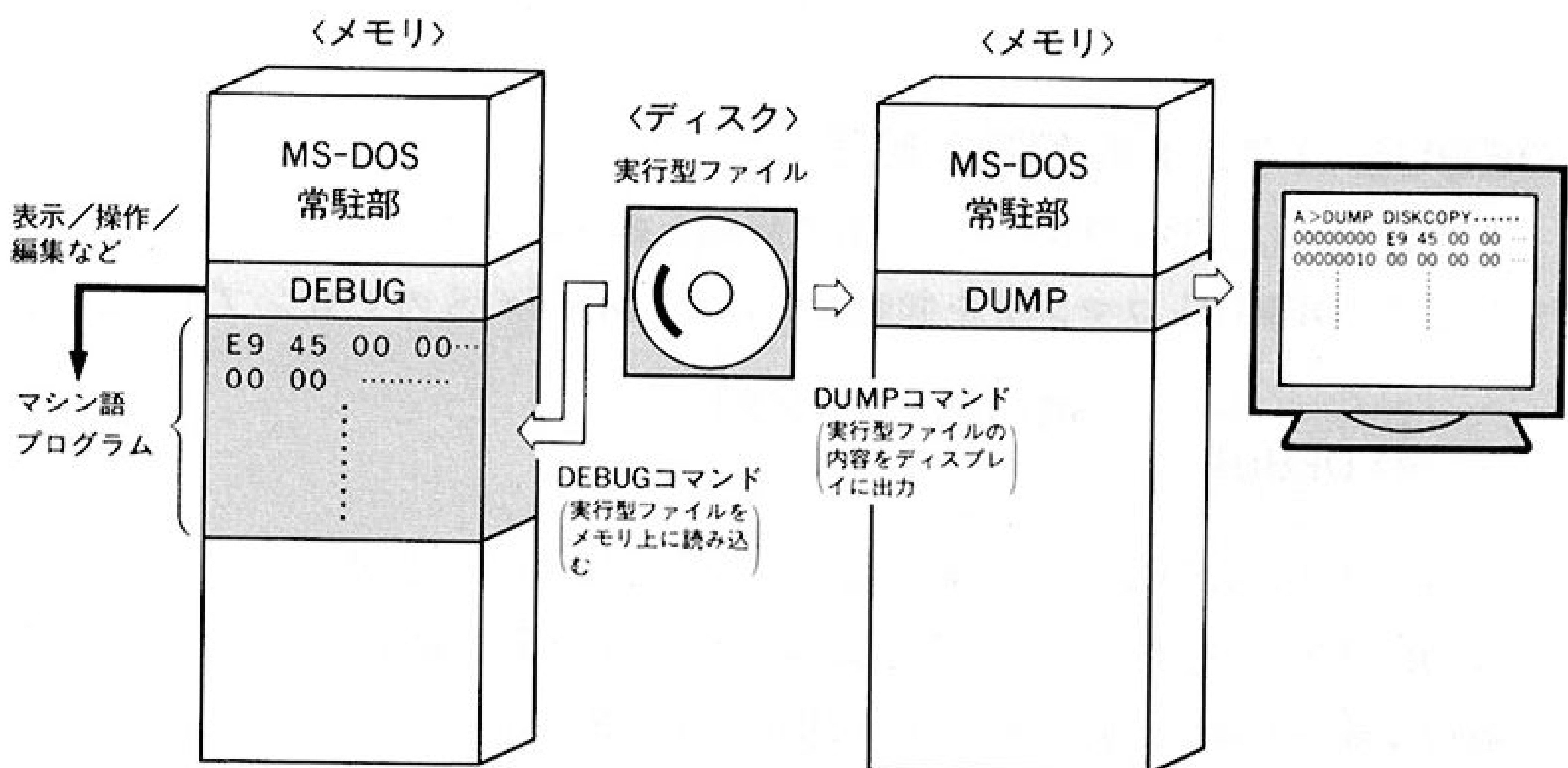


図3-1　DEBUGコマンドとDUMPコマンドの違い

---

*「SYMDEB」コマンドを使っても本書の実習はまったく同じように行うことができる．なお，DEBUGコマンドやSYMDEBコマンドがMS-DOSのシステムディスクには含まれず別売されている機種もある．

DEBUG コマンドは非常に豊富な機能を持っていますが，ここではマシン語プログラムを操作するために最低限必要な以下の4つのコマンドについて説明します．

コマンド名	機　　能
Dコマンド	メモリの内容を表示する(Dump（ダンプ）)
Eコマンド	メモリの内容を変更する(Enter（エンター）)
Wコマンド	メモリの内容をファイルに書き込む(Write（ライト）)
Qコマンド	DEBUGを終了してMS-DOSに戻る(Quit（クイット）)

DEBUGに入力するコマンドは大文字／小文字のどちらでもかまわない

表 3-1　3章で使用する DEBUG のコマンド

このなかでDコマンドとEコマンドは，4章以降で何度も使うことになりますから，その使い方をしっかり覚えてください．これ以外の「DEBUG」コマンドの機能については，後の章で追々紹介していきます．

DEBUGコマンドの起動と終了

まずはじめに，DEBUG コマンドの起動と終了の方法を見ていくことにしましょう．DEBUG コマンドを起動するには，MS-DOS のプロンプト* から，

```
A> DEBUG ⏎
```
（A> ← MS-DOS のプロンプト）

と入力します．DEBUG が起動するとプロンプトが「-」に変わり，この状態では DEBUG に対するコマンドしか受け付けなくなります．

最初に知っておく必要があるのが，DEBUG を終了するコマンドです．DEBUG を終了するには，DEBUG の「-」のプロンプトに対し「Q」(Quit) を入力します．すると DEBUG を終了し，もとの MS-DOS のプロンプトに戻ります(図 3-2)．

* MS-DOS のプロンプトとは「A>」のこと．プロンプトには入力をうながすという意味があり，コマンドを受け付ける準備ができていることを表している．

```
A>DEBUG↵ ……………DEBUGコマンドを起動する
-Q↵ ……………………QコマンドでDEBUGを終了し，MS-DOSに戻る
:……………DEBUGのプロンプト
A>
```

図 3-2　DEBUG コマンドの起動と終了

また DEBUG コマンドを起動するには，DUMP コマンドと同じようにファイル名を指定して起動する方法もあります．これは MS-DOS のプロンプトから，

A> DEBUG ファイル名↵

として起動します．これにより DEBUG コマンドは，指定したファイルをメモリ上に読み込んでからプロンプト「-」を表示します．この章では，実行型ファイルを読み込んでその内容を操作するので，起動時にファイル名を指定する方法をとります．



Dコマンド（メモリのダンプ）

Dコマンドは，メモリの内容を見る（ダンプする）ためのコマンドです．このDコマンドは基本的にはDUMPコマンドと同じことをしますが，DUMPコマンドがディスク上のファイルの内容をダンプするのに対して，DEBUGのDコマンドはメモリの内容をダンプするという点が異なります．したがってファイルの内容をダンプするには，DEBUGの起動時にファイル名を指定してあらかじめファイルの内容をメモリに読み込んでおけばよいのです．その状態でDコマンドを入力することによって，DUMPコマンドと同じ結果を得ることができます．

Dコマンドでは，調べたいメモリの範囲をアドレス（番地）で指定することができます．Dコマンドの使い方には，次に示す3通りがあります．

D XXXX YYYY ↵	…… XXXX$_{\mathrm{H}}$番地からYYYY$_{\mathrm{H}}$番地までをダンプする．
D XXXX ↵	…… XXXX$_{\mathrm{H}}$番地から128$_{\mathrm{H}}$番地分をダンプする．
D ↵	…… 1つ前に実行したDコマンドの最終番地（1番始めに実行する場合は0100$_{\mathrm{H}}$番地）から，128$_{\mathrm{H}}$番地分をダンプする．

（下線部を入力）

ここでXXXX$_{\mathrm{H}}$やYYYY$_{\mathrm{H}}$として指定する値がアドレスになります．アドレスは4桁の16進数として指定しますが，この読み方や数え方は「4.2　2進数と16進数」でくわしく解説します*．ここでは，メモリの位置を指定するための背番号だと思ってください．

それでは，実際に2章でやった「DISKCOPY.COM」ファイルをメモリ上に読み込んで，Dコマンドを実行してみましょう（図3-3）．

DUMPコマンドを実行したときと同じような結果が表示されました．若干違うところがありますが，それは左側のアドレスの部分が2つの4桁の数字の組に分かれていて，「：」（コロン）で区切られていることです．これはともにアドレスを表し，コロンの左4桁を「セグメントアドレス」，右4桁を「オ

＊ 16進数は10進数と区別するために，終わりに「H」（Hexadecimal notation：16進表示）を付ける．

フセットアドレス」と呼ぶのですが，くわしくは5.5章で解説します．この章では右4桁のことを単にアドレスと呼ぶことにします．

各行には数字とアルファベットの組が16個並んでいます．この各々が1バイトのメモリの値を16進数とキャラクタ（文字）で表したものになります*．

```
セグメントアドレス……システムによってこの値は異なる．くわしくは5.5章で解説
(オフセット)アドレス…この章でいうアドレスとはこの部分を指す
                                                              左側のデータに対応す
                                                              る文字が表示される
A>DEBUG DISKCOPY.COM ↵ …DEBUGを起動し，「DISKCOPY.COM」ファイルをメモリ上に読み込む
-D↵ ………Dコマンドを実行する（1番始めに実行されたので0100H番地から128バイト分表示する）
3651:Ø1ØØ  E9 45 Ø4 ØØ ØØ ØØ ØØ ØØ-ØØ 2F Ø1 56 ØØ ØØ ØØ ØØ   iE......./.V....
3651:Ø11Ø  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   ................
3651:Ø12Ø  ØØ ØØ ØØ ØD ØA 44 49-53 4B 43 4F 5Ø 59 2Ø 76   .....DISKCOPY v
3651:Ø13Ø  65 72 73 69 6F 6E 2Ø 32-2E 31 ØØ ØD ØA 82 63 82   ersion 2.1....c.
3651:Ø14Ø  6E 82 72 82 CC 83 6F 81-5B 83 57 83 87 83 93 82   n.r.L.o.[.W.....
3651:Ø15Ø  88 E1 82 A2 82 DC 82-B7 ØØ ØD ØA 91 97 82 E8   k.a.".¥.7......h
3651:Ø16Ø  A4 83 66 83 42 83 58-83 4E 82 FØ 83 68 83 89   .$.f.B.X.N.p.h..
3651:Ø17Ø  83 43 83 75 2Ø 4Ø 3A 2Ø-82 C9 91 7D 93 FC 82 B5   .C.u @: .|.}.:.5
   (0100H番地 ～ 010FH番地 ／ 0170H番地 ～ 017FH番地)
-D↵ ……………続けてDコマンドを実行する（先にダンプした最終番地の次の番地から128バイト分表示する）
3651:Ø18Ø  82 C4 82 AD 82 BE 82 B3-82 A2 ØØ ØD ØA 8E F3 82   .D.-.>.3."....s.
3651:Ø19Ø  F 91 A4 83 66 83 42 83-58 83 4E 82 FØ 83 68 83   /.$.f.B.X.N.p.h.
3651:Ø1AØ  C9 83 43 83 75 2Ø 4Ø 3A-2Ø 82 C9 91 7D 93 FC 82   ..C.u @: .|.}.:.
3651:Ø1BØ  B5 82 C4 82 AD 82 BE 82-B3 82 A2 ØD ØA 8F 8Ø 94   5.D.-.>.3.".....
3651:Ø1CØ  F5 82 AA 82 C5 82 AB 82-BD 82 E7 82 C7 82 EA 82   u.*.E.+.=.g.G.j.
3651:Ø1DØ  A9 82 CC 83 4C 81 5B 82-FØ 89 9F 82 B5 82 C4 82   ).L.L.[.p...5.D.
3651:Ø1EØ  AD 82 BE 82 B3 82 A2 2Ø-ØØ ØD ØA ØD ØA 83 66 83   -.>.3." ......f.
3651:Ø1FØ  42 83 58 83 4E 82 CC 83-52 83 73 81 5B 82 FØ 8D   B.X.N.L.R.s.[.p.
   (16バイト ／ 8行)
-D Ø1ØØ↵ ……………0100H番地から128バイト分を再びダンプする
3651:Ø1ØØ  E9 45 Ø4 ØØ ØØ ØØ ØØ ØØ-ØØ 2F Ø1 56 ØØ ØØ ØØ ØØ   iE......./.V....
3651:Ø11Ø  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   ................
3651:Ø12Ø  ØØ ØØ ØØ ØØ ØD ØA 44 49-53 4B 43 4F 5Ø 59 2Ø 76   ......DISKCOPY v
3651:Ø13Ø  65 72 73 69 6F 6E 2Ø 32-2E 31 ØØ ØD ØA 82 63 82   ersion 2.1....c.
3651:Ø14Ø  6E 82 72 82 CC 83 6F 81-5B 83 57 83 87 83 93 82   n.r.L.o.[.W.....
3651:Ø15Ø  AA 88 E1 82 A2 82 DC 82-B7 ØØ ØD ØA 91 97 82 E8   *.a.".¥.7......h
3651:Ø16Ø  91 A4 83 66 83 42 83 58-83 4E 82 FØ 83 68 83 89   .$.f.B.X.N.p.h..
3651:Ø17Ø  83 43 83 75 2Ø 4Ø 3A 2Ø-82 C9 91 7D 93 FC 82 B5   .C.u @: .|.}.:.5
-D Ø12Ø Ø12F↵ ……………0120H番地から012FH番地まで16バイト分ダンプする
3651:Ø12Ø  ØØ ØØ ØØ ØØ ØD ØA 44 49-53 4B 43 4F 5Ø 59 2Ø 76   ......DISKCOPY v
-D Ø13Ø Ø137↵ ……………0130H番地から0137H番地まで8バイト分ダンプする
3651:Ø13Ø  65 72 73 69 6F 6E 2Ø 32                          ersion 2
-Q↵ ……………QコマンドでDEBUGを終了する

A>
        〈注意〉使用機種によって，このデータは異なります．
```

図 3-3　Dコマンドの実行例

＊各行の右側にある文字による表示は，DUMP コマンドによるものと一部異なる．その理由は202ページの注を参照．

Eコマンド（メモリ内容の変更）

Eコマンドは，指定したアドレスのメモリの内容を任意の16進数の値に書き換えるコマンドです．これまで見てきたように，メモリは1バイトという単位で読むことができ，書き込みもやはり1バイト単位に行います．アドレスや16進数について，ここでは理解している必要はありません．とりあえずコマンドの使い方を覚えてください．

Eコマンドの使い方にはいくつか方法がありますが，まずその1つを紹介しましょう．

E XXXX⏎　……　アドレスXXXXH番地からメモリの内容を変更するモードに入る．

○○○○：XXXX　●●．■■［スペース］

アドレスXXXXH番地とその内容「●●」が表示されるので，変更しようとする値「■■」を入力しスペースキーを押す．

○○○○：XXXX　●●．■■　●●．■■［スペース］

次のアドレスのメモリの内容「●●」が表示されるので，変更しようとする値「■■」を入力しスペースキーを押す．

⋮

以下，同じようにして連続したメモリの内容を修正できる．

⋮

○○○○：XXXX　●●．■■　●●．■■　……　●●．■■⏎

リターンキーを押すと，DEBUGのコマンドモードに戻る．

（■■は，2桁の16進数を指定する．）

このように，この方法では1バイトのメモリの内容を確認しながら，その値を変更していくことができます．

メモリの内容を次々と変更する場合は，スペースキーで次のアドレスへ移動するという点に注意してください．リターンキーを押すと，メモリ内容の変更はすべて終了したという指定になりDEBUGのプロンプト「-」に戻ってしまいます．

次の図3-4にこの実行例を示しますが，図では次々とアドレスが進んでいく様子がうまく示せませんので，自分で試してみてください．

```
A>DEBUG DISKCOPY.COM↵
-D Ø1ØØ Ø1ØF↵ ··············0100H番地から010FH番地までのメモリの内容をダンプする
3651:Ø1ØØ  E9 45 Ø4 ØØ ØØ ØØ ØØ ØØ-ØØ 2F Ø1 56 ØØ ØØ ØØ ØØ   iE......./.V....
-E Ø1ØØ↵ ·····0100H番地からメモリの内容を変更する　8個入力すると自動的に改行され，次のアドレスに進む
3651:Ø1ØØ  E9.31[SP]45.32[SP]Ø4.33[SP]ØØ.[SP]ØØ.35[SP]ØØ.[SP]ØØ.37[SP]ØØ.38[SP]
3651:Ø1Ø8  ØØ.39↵ ·····リターンキーで変更終了　　スペースキーのみを押すと，データを変更せずに
-D Ø1ØØ Ø1ØF↵ ··············変更されたメモリの内容を確認する　次のアドレスに進む
3651:Ø1ØØ  [31] [32] [33] ØØ [35] ØØ [37] [38]-[39] 2F Ø1 56 ØØ ØØ ØØ ØØ   123.5.789/.V....
-Q↵            メモリの内容が変更された

A>                                    [SP] : スペースキー
```

図3-4　Eコマンドの実行例（1）

上に挙げた方法は，現在の値を確認しながらデータを変更できるのでたいへん便利ですが，その必要がない場合には面倒なこともあります．Eコマンドではアドレスとともに書き換えたい値を指定することによって，そのアドレスから始まる連続したメモリの内容を変更することもできます．

その使い方を次に示します．

> **E XXXX ■■, ■■, ■■, ……, ■■↵**
>
> アドレスXXXX(H)番地から始まるメモリの内容を「■■　■■　■■　……　■■」に変更する．
>
> （■■は，2桁の16進数を指定する．）

変更するデータの並びは，「,」で区切って指定します．この実行例は，次の図3-5のようになります．

```
A>DEBUG DISKCOPY.COM↵
-D Ø1ØØ Ø1ØF↵ ··············データを変更する前のメモリの値を確認する
3651:Ø1ØØ  E9 45 Ø4 ØØ ØØ ØØ ØØ ØØ-ØØ 2F Ø1 56 ØØ ØØ ØØ ØØ   iE......./.V....
-E Ø1ØØ 41,42,43,44↵ ··············0100H番地から連続するメモリの値を「41 42 43 44」に変更する
-D Ø1ØØ Ø1ØF↵ ··············変更結果を確認する
3651:Ø1ØØ  [41] [42] [43] [44] ØØ ØØ ØØ ØØ-ØØ 2F Ø1 56 ØØ ØØ ØØ ØØ   ABCD..../.V....
-Q↵

A>
```

図3-5　Eコマンドの実行例（2）

さらにこの形式の応用として，「■■」で指定した16進数のかわりに文字列を指定することができます．これは次節で取り上げるマシン語プログラムのメッセージを変更する際に用います．

文字列は「'」（シングルクォーテーション）* で囲んで指定することによって，対応する16進数のキャラクタコード（273 ページのキャラクタコード表を参照）に自動的に変換されるのです．

これは，次のような書式で指定します．

E XXXX '□□□□' ↵

アドレス XXXXH番地から始まるメモリの内容を文字列「□□□□」に対応するキャラクタコードに変更する．

図 3-5 と同じことを，この方法でやってみます（図 3-6）．

```
A>DEBUG DISKCOPY.COM↵
-D 0100 010F↵
3651:0100  E9 45 04 00 00 00 00 00-00 2F 01 56 00 00 00 00   iE......./.V....
-E 0100 "ABCD"↵ ………………アドレス0100H番地から始まるメモリの内容を文字列「ABCD」に対応する
-D 0100 010F↵              キャラクタコードに変更する
3651:0100  41 42 43 44 00 00 00 00-00 2F 01 56 00 00 00 00   ABCD...../.V....
-Q↵        ↑  ↑  ↑  ↑    対応するキャラクタコードに変更された
           A  B  C  D
A>
```

図 3-6　E コマンドの実行例（3）

* 「"」（ダブルクォーテーション）を使うこともできる．

Wコマンド（メモリ内容のファイルへの書き込み）

メモリの内容をディスク上のファイルに書き込みます．これはDEBUGコマンドでファイルをメモリに読み込んで，その内容を変更したものを書き戻したい場合に使用します．書き込むファイル名は，DEBUGを起動した時に指定したファイル名と同じです．このコマンドは，「w」と入力するだけで実行されます（図3-7）．

このコマンドを実験する場合には，必ずバックアップをとったディスクで行ってください．キー入力ミスなどでディスクの内容を破壊する可能性もありますから注意が必要です．

```
A>DIR DISKCOPY.COM↵

 ドライブ A: のディスクのボリュームラベルはありません
 ディレクトリは A:¥

DISKCOPY COM      6880  85-05-19  20:44
          1 個のファイルがあります
     44032 バイトが使用可能です

A>DEBUG DISKCOPY.COM↵
-W↵ ……………Wコマンドを実行し，読み込んだマシン語プログラムをそのままディスクに書き戻す
Writing 1AE0 bytes（ここでは，ファイルの内容を壊さないようにするために，メモリの内容を変更しない）
-Q↵
   ……………書き込んだバイト数が表示される．この値はファイルサイズを16進数で表したもの
A>DIR DISKCOPY.COM↵……新たに書き込みが行われた「DISKCOPY.COM」ファイルを確認する

 ドライブ A: のディスクのボリュームラベルはありません
 ディレクトリは A:¥

DISKCOPY COM      6880  86-12-18  16:40……………新たに書き込みが行われたので，
          1 個のファイルがあります                   日付／時刻が変わっている
     44032 バイトが使用可能です

A>
```

図3-7　Wコマンドの実行例

3.2 メッセージの変更

前節でDEBUGコマンドの基本的な使い方を解説しました．そこでここではDEBUGコマンドを使って，メモリに読み込んだマシン語ファイルを実際に操作してみましょう．ここでは，2章でダンプしてその内容を確認した「DISKCOPY.COM」ファイルのメッセージ部分を変更してみることにします．

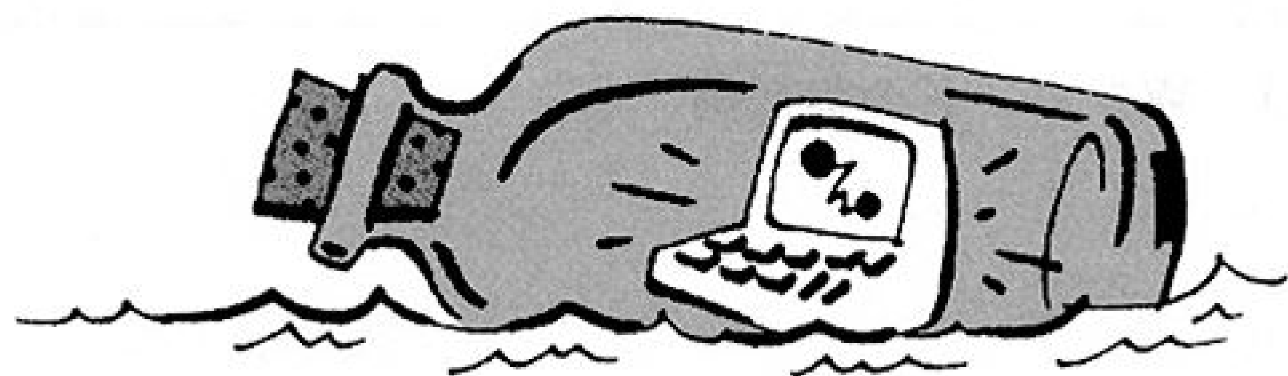
この実験は簡単に行えますから，ぜひ自分でやってみてください．DEBUGの操作を覚えるという意味でも，コンピュータの扱うマシン語を理解するという意味でも自分で体験することは欠かせません．

なお，この実験では，ディスク上のファイルを書き換えてしまうので，必ずバックアップをとったディスクで行います．

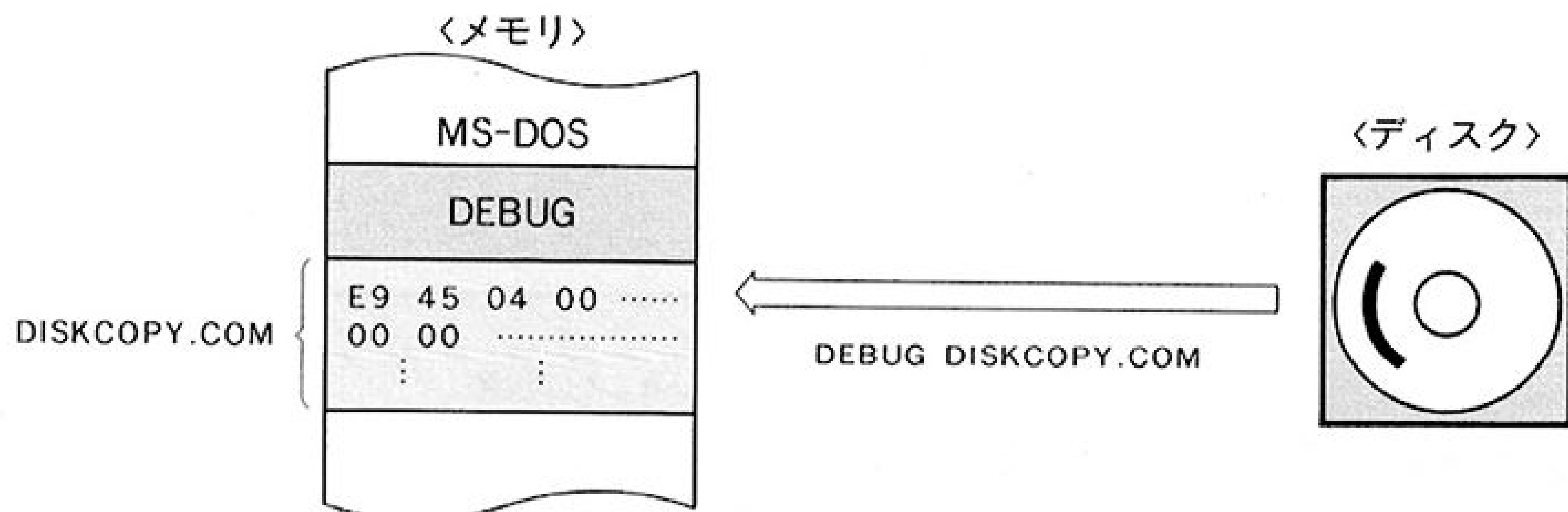
英文のメッセージを変更する

2章でダンプしたときにわかったように，DISKCOPYコマンドを実行したときに表示される「DISKCOPY version 2.1」というメッセージが，DISKCOPY.COMファイルの中にそのままのかたちでありました．これをDEBUGコマンドで書き換えてしまえば，メッセージが変わってしまうのではないでしょうか？　そこでこのメッセージを書き換える実験を行います．

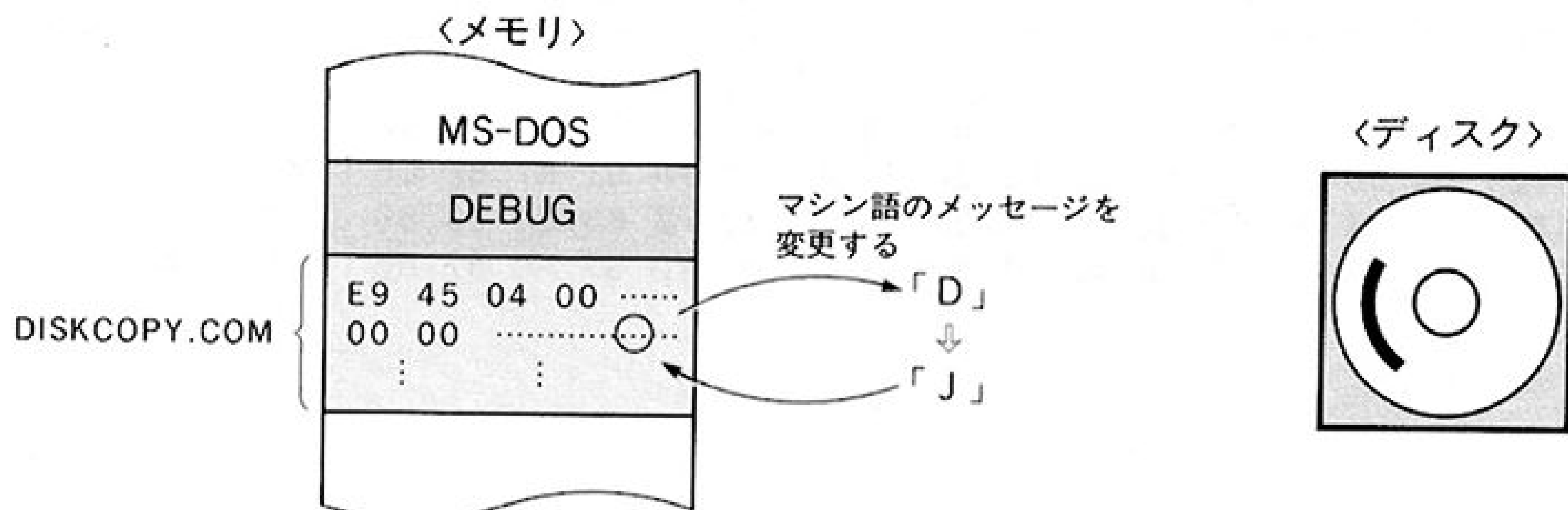
ファイル中のデータを書き換える手順は次の通りです（図3-8）．



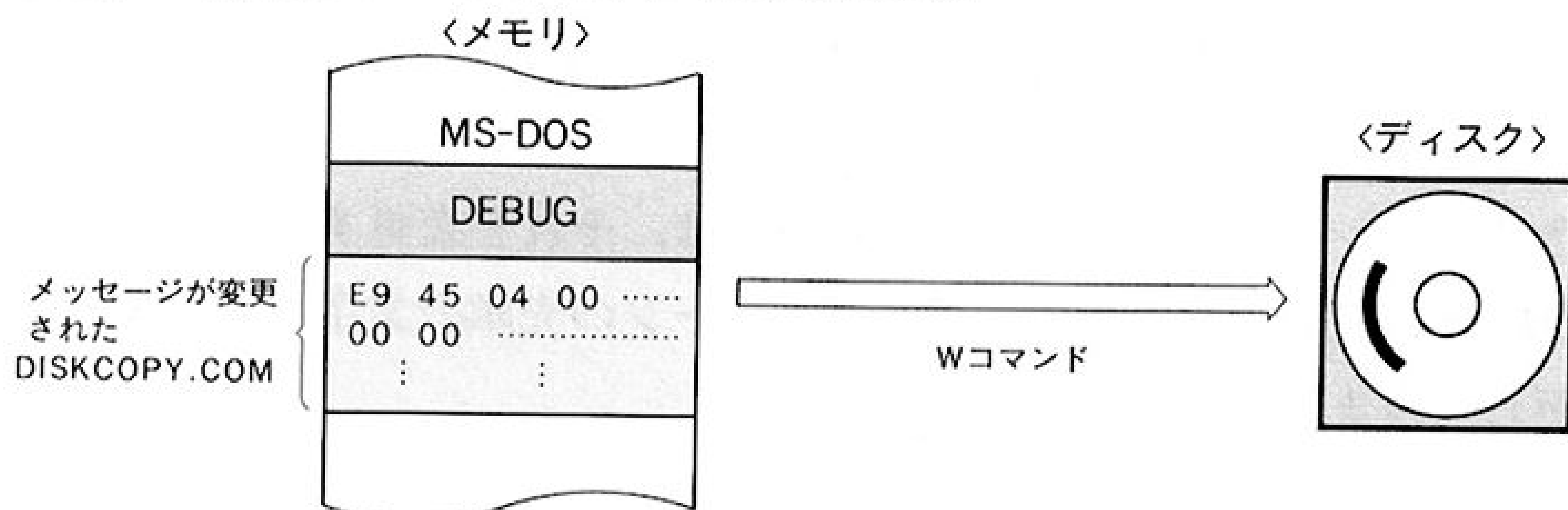
① DEBUGを起動してファイルを読み込む



② Eコマンドでメッセージを書き換える



③ 変更した結果をWコマンドでファイルに書き込む



④ QコマンドでDEBUGを終了する



図3-8　マシン語ファイルのメッセージを変更する手順

それでは実験を始めましょう．まず，DEBUG を起動して「DISKCOPY.COM」ファイルを読み込み，「DISKCOPY version 2.1」のメッセージをDコマンドで確認してみます*．



図 3-9 Dコマンドでメッセージを確認する

ここで DISKCOPY.COM のアドレスが，DUMP コマンドと違って 0100_H 番地から始まっているという点に注意しておいてください．

さて，メッセージのアドレスがわかれば，それを変更するのは簡単です．DEBUG のEコマンドを使って，メッセージの先頭の文字「D」を「J」に変更してみましょう．

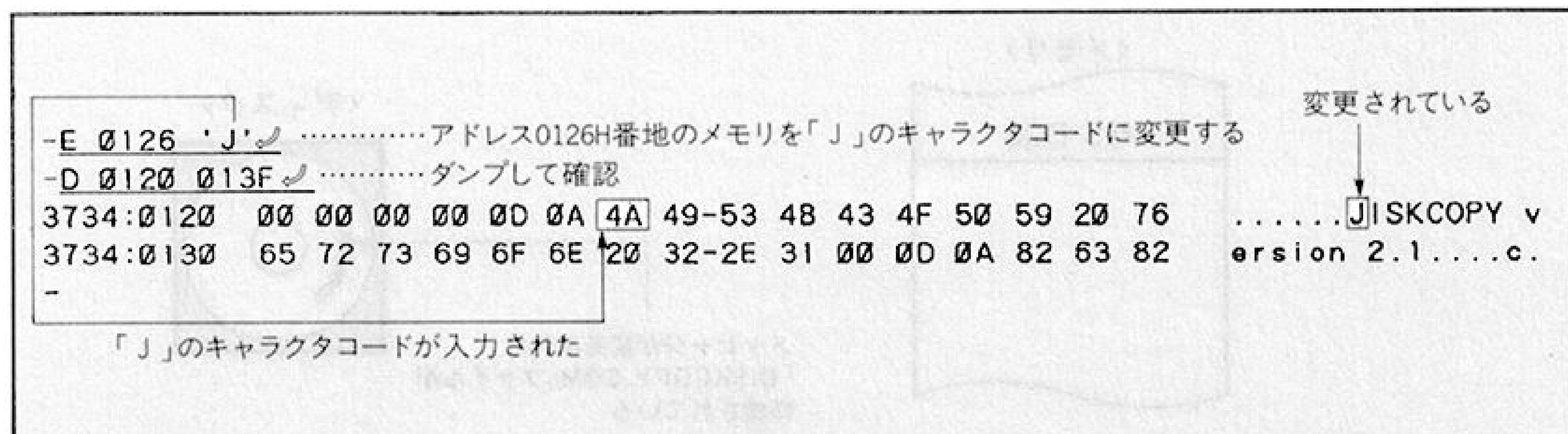


図 3-10 Eコマンドでメッセージを変更する

＊富士通の FM 系のマシンでは，このメッセージが全角で表示される．この場合は，次項の「日本語のメッセージを変更する」を参照のこと．

図3-10のように変更がうまくいったことが確認できたら，Wコマンドを使ってもとのファイルに変更したものを書き込み，QコマンドでDEBUGを終了します（図3-11）．

```
-W↵ ……………メッセージが変更されていることを確認したら，Wコマンドでディスクに書き戻す
Writing 1AEØ bytes
-Q↵ ……………DEBUGを終了し，MS-DOSに戻る

A>
```

図3-11　Wコマンドで変更したメモリの内容をファイルに書き戻す

それでは，メッセージを変更したDISKCOPYコマンドを実行してみましょう．実行する際はディスクドライブには実験用ディスクだけを入れて，大事なディスクは絶対に入れておかないようにしてください．また，ディスクのコピーを実際に行うことが目的ではないので，バージョンナンバーを確認したら，すぐに[CTRL]+[C]を入力してDISKCOPYコマンドを中止しましょう．

```
A>DISKCOPY A: B:↵ ……………メッセージを変更したDISKCOPYコマンドを実行してみる

JISKCOPY version 2.1……………メッセージが変わった！

ディスクのコピーを行います

送り側ディスクをドライブ A: に挿入してください
受け側ディスクをドライブ B: に挿入してください
準備ができたらどれかのキーを押してください ^C……………[CTRL]+[C]で途中終了する

A>
```

図3-12　メッセージを変更したDISKCOPYコマンドを起動する

もし，うまくいかなかった人は，もう一度挑戦してみてください．その場合は前回の実験によって，ファイルの内容のどこかが壊れているかもしれないので，必ずシステムディスクからもう一度ファイルをコピーし直してから実験します．決してむずかしくはないはずです．

このようにDEBUGコマンドを使えば，テキストファイル以外の実行型ファイルについてもメッセージなどのマシン語データを変更できることがわかったでしょう．ここで忘れないでほしいことがあります．マシン語プログラムには，純粋にプログラムの部分とメッセージなどのデータの部分があることを2章で述べましたが，このように変更してよいのはメッセージの部分だけです．プログラムの部分を勝手に変更すると，実行時に確実に暴走してしまいますから注意が必要です．

●

漢字のメッセージを変更する

今度は漢字のメッセージを変更してみましょう．漢字のメッセージも基本的には，さきほどの半角の英数字のメッセージと同様の方法で変更することができます．

ここでは「送り側ディスクをドライブＡ：に挿入してください」というメッセージ*を，バージョンナンバーと同じように書き換えることによって「転送元ディスクを……」というメッセージに変更してみます．

メッセージを変更する際に気をつけなければならないのが，変更する部分の文字数です．変更する文字数が変更前と変更後で一致していなければ，正しい文字列にならないばかりか，必要なデータを破壊してしまうことにもなりかねません．変更する文字列は必ず同じ長さになるようにしてください．

それでは実験に入りましょう．まず，変更する文字列のアドレスを調べます．DEBUGのDコマンドでは漢字を表示することができませんから，前節のCDUMPコマンドを使ってアドレスを求めます．次の図3-13は，2章でCDUMPコマンドにより「DISKCOPY.COM」ファイルをダンプしたリスト（図2-8）の一部です．

「送り側ディスクを……」というメッセージは2行目のなかほどにあります．これがこの行の先頭から何文字目にあるか数えてみましょう．漢字や全角文字は2文字と数えることに注意してください．「送り側……」のメッセージは先頭から，28文字目になります．

---

*機種によって，このメッセージは異なる場合があります．

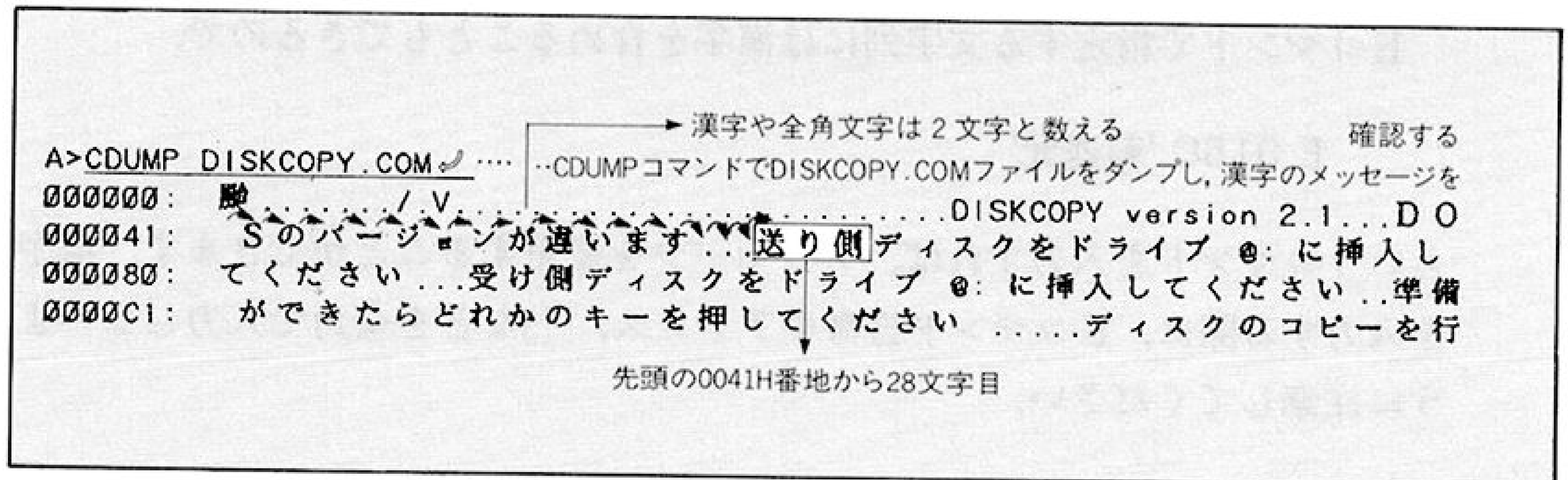


図 3-13　CDUMP コマンドによる漢字メッセージの確認

さて，これがわかったら今度はDEBUGコマンドを起動して「DISKCOPY.COM」を読み込み，Dコマンドでメモリの中身を表示させてみます(図3-14).

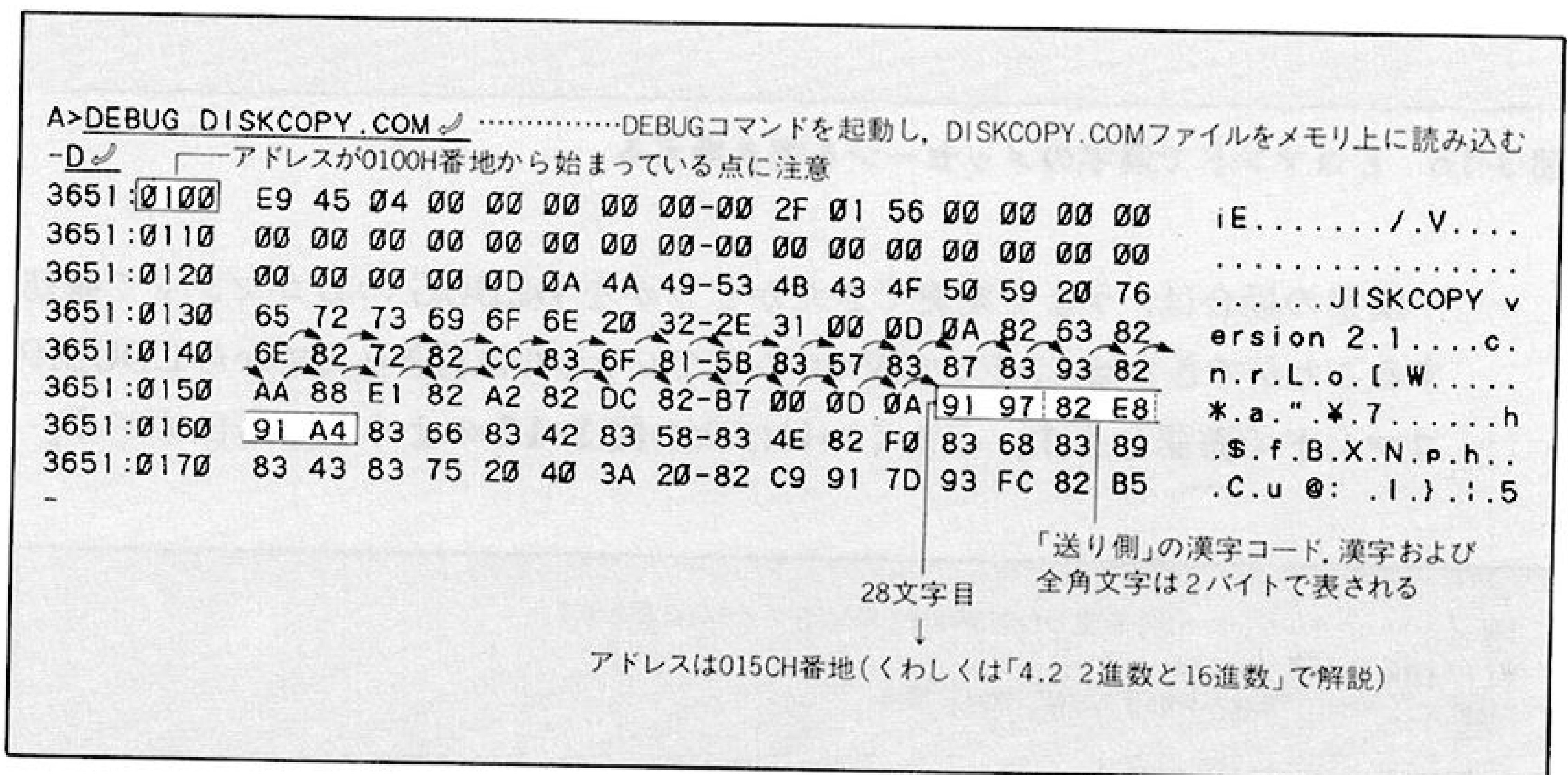


図 3-14　Dコマンドでメッセージのアドレスを確認

ここでDEBUGでは，アドレス0100_H番地からファイルが読み込まれることに気をつけると，アドレス0141_H番地から28番目が「送り側ディスク……」の漢字のメッセージの先頭になります．このアドレスは「4.2　2進数と16進数」で解説しますが，16進数で表すと「015C」になります*.

＊富士通のFM系のマシンではDISKCOPYコマンドのバージョンが3.11版の場合「0625」，3.20版の場合「05C3」となる．

Eコマンドで指定する文字列には漢字を含めることもできるので,

E 015C '転送元'

というコマンドを入力すれば,メッセージを変更することができます.漢字を入力する際に,Eコマンド自身やアドレス,「'」などを全角で入力しないように注意してください.

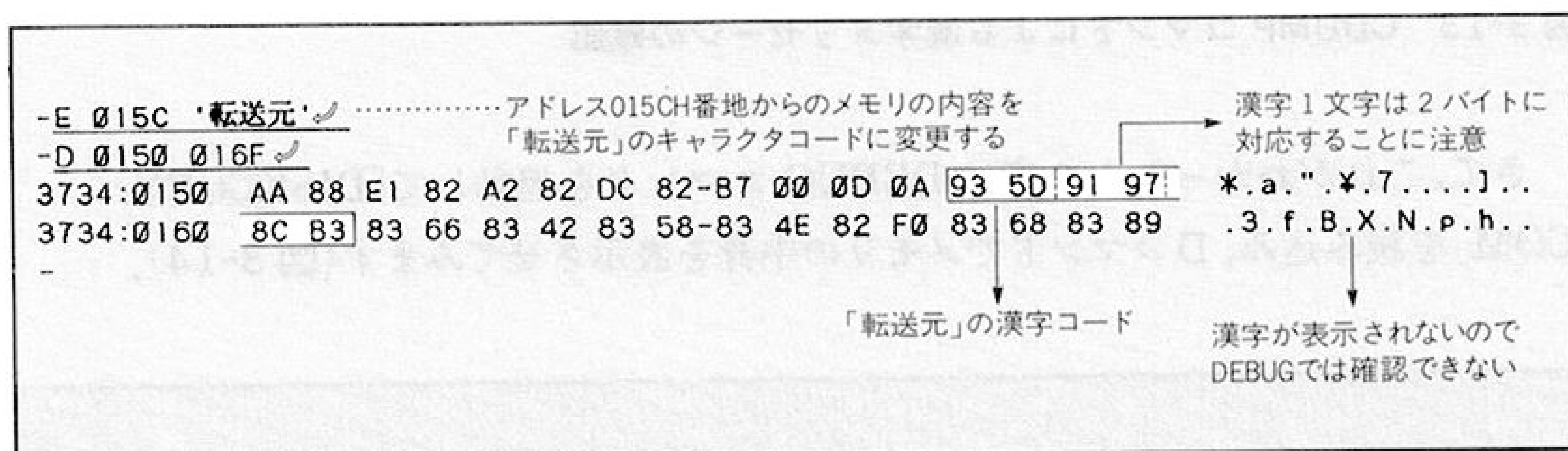


図 3-15　Eコマンドで漢字のメッセージを書き換える

漢字の場合は,うまく変更できたかどうかをDEBUGのDコマンドで確認することができません.そこでWコマンドでファイルに書き込んでからCDUMPコマンドで確認します.うまくいけば次の図3-16のようになるはずです.

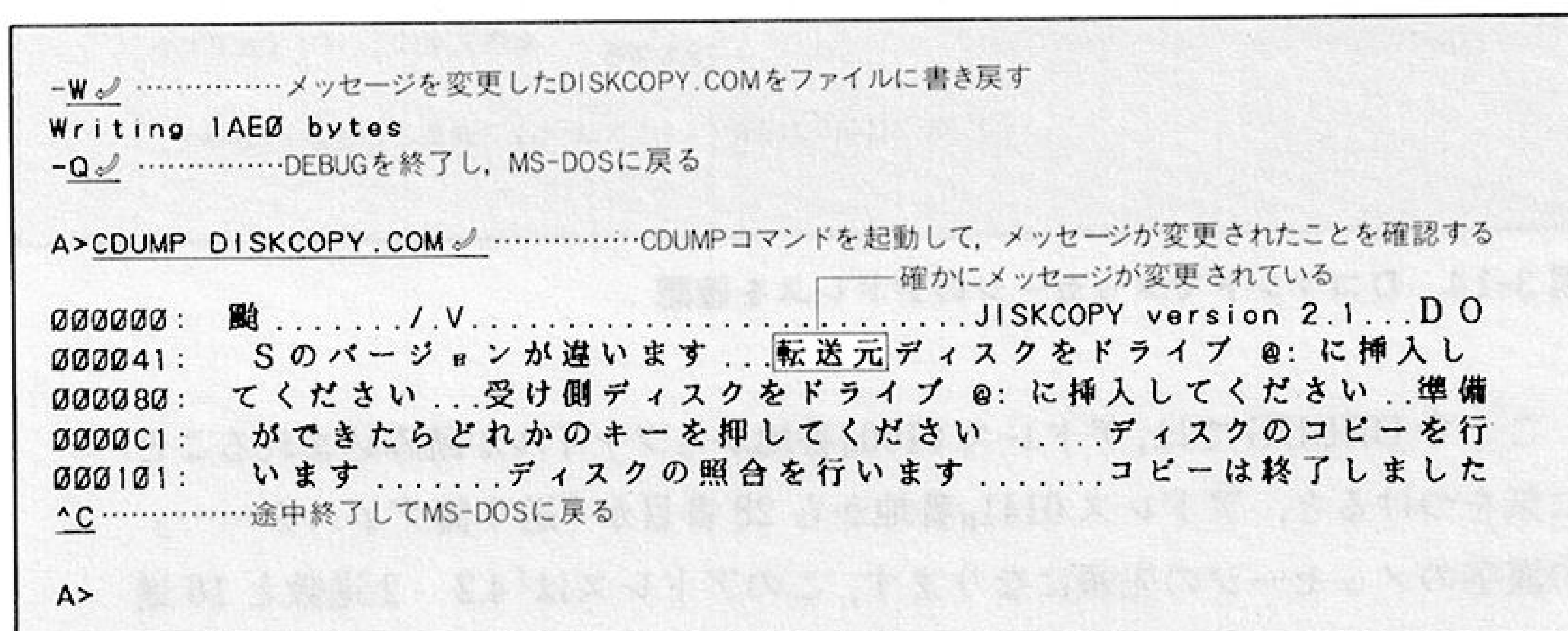


図 3-16　Wコマンドでファイルに書き込み,CDUMPコマンドで確認する

もし失敗していれば，システムディスクからDISKCOPY.COMファイルをコピーし直して，もう一度始めからやってみてください．

では，メッセージを変更したDISKCOPYコマンドを実行してみましょう．今度もやはりディスクのコピーが目的ではないので，メッセージを確認したらすぐに[CTRL]+[C]で中止してください．次の図3-17のように変更されたメッセージが表示されるはずです．

```
A>DISKCOPY A: B:↵ ……………漢字のメッセージを変更したDISKCOPYコマンドを実行する

JISKCOPY version 2.1

ディスクのコピーを行います
        漢字のメッセージも変わった！
[転送元]ディスクをドライブ A: に挿入してください
受け側ディスクをドライブ B: に挿入してください
準備ができたらどれかのキーを押してください ^C ……………途中終了してMS-DOSに戻る

A>
```

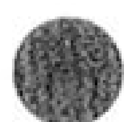
図3-17　漢字のメッセージを変更したDISKCOPYコマンドを実行する

以上でDEBUGコマンドを使ってマシン語データを変更する実験を終わりますが，マシン語プログラム中のメッセージの確認や変更は簡単にできることがわかってもらえたでしょうか．この章だけでは16進数やアドレスといった言葉がわからないので，呪文のように思えるかもしれません．しかし，それらは以後の章でやさしく解説していますので，読み進んでいくにつれて本章でやったことの意味がわかってくるでしょう．

この章ではDEBUGコマンドを使ってファイルをメモリに読み込み，その「メモリ内容の確認や変更ができた」ことが重要なのです．

4

これだけは覚えて欲しいコンピュータの知識



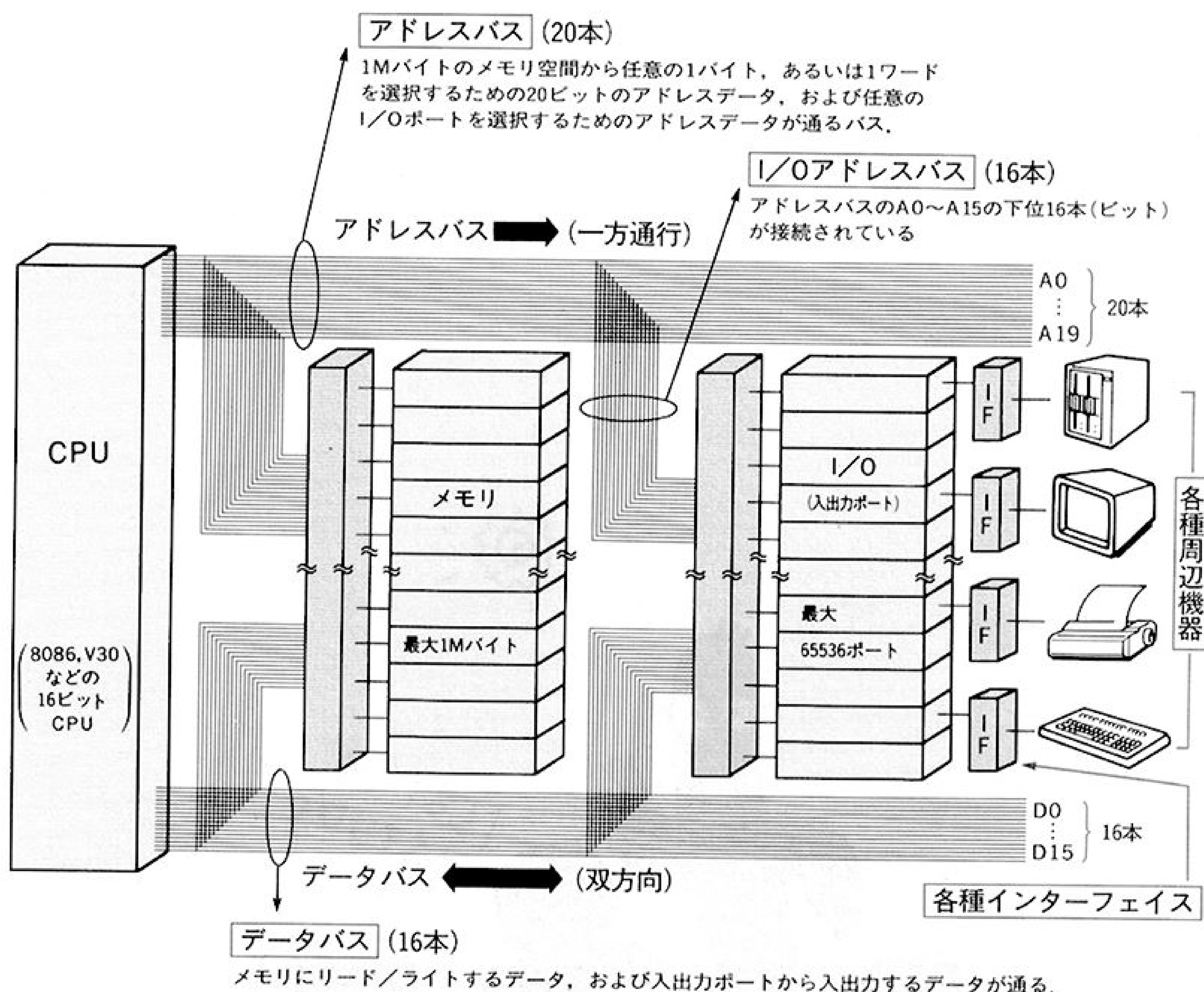
4

マシン語はコンピュータ・システムの仕組みに直接かかわるものであり，決して切り離して考えることはできません．この章ではマシン語を理解するために必要な，コンピュータ・システムについての基礎的な知識を解説します．

ここで解説するのはパーソナルコンピュータのカタログ等によく出てくる，CPU，メモリ，16 ビット，256K バイト，クロック，バスなどの言葉の概念です．すでにこれらの意味が理解できる方は読み飛ばしてもらってもかまいません．言葉は知っているものの，その意味するところがあまりよくわからないという方は，この章をよく読んで理解してください．

4.1　コンピュータ・システムの構成

一般的な16ビットパーソナルコンピュータのシステム構成について解説しましょう．コンピュータ・システムは，基本的には次の図 **4-1** のように構成されています．



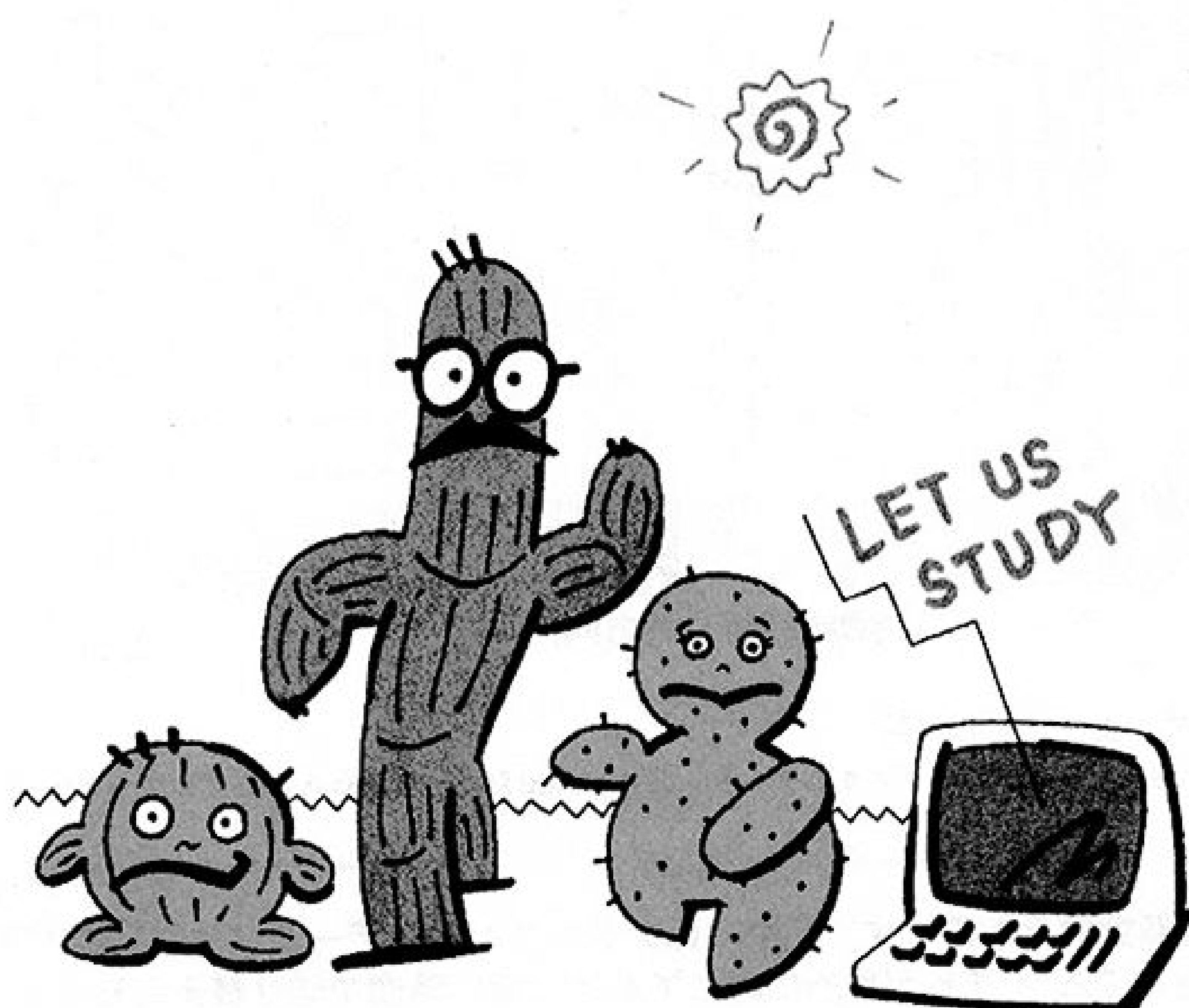
注意！

この図はあくまで概念的なものです．実際にはアドレスバスとデータバスは別々ではなく，同じ信号線を共有しており，それが目的に応じて非常に短い時間で切り替えられます．しかし，プログラミングの立場からは，この図のように考えてもまったくさしつかえはなく，逆にこの方が理解しやすいのです．したがって以降の図や説明も，この図にあるような概念で示します．

図 4-1　コンピュータ・システムの構成

この図で示されるように，コンピュータ・システムは，「CPU」，「メモリ」，「I/O」とそれにつながる各種周辺機器，データが行き来する「バス」と呼ばれる信号線などの基本的なブロックの組合せでしかないのです．この図はコンピュータの最も基本的な構成図であり，これらのブロックの関係をよく見ておいてください．

もちろん，この図には示されていない各種の制御信号が通る信号線もありますが，コンピュータ・システムの基本的な構成はこの図で十分表されています．なお，この図は，後の節を読み進めていく際にも理解の助けとなりますから，必要に応じて参照するとよいでしょう．



4.2　2進数と16進数

コンピュータの中身はすべて「0」か「1」のどちらかしかない2進数の世界です．どんなに複雑な処理であってもコンピュータの内部では，この「0」と「1」の2つの値だけで処理が行われており，「0」と「1」をいくつも組み合わせることでさまざまな情報を表現します．この「0」か「1」かという2つの状態しか持たない情報量のことを，**ビット**（Bit）と呼びます．ビットの概念はコンピュータのアーキテクチャ*の根幹をなすものであり，コンピュータを理解する上での最も重要なキーポイントでもあります．また，いくつものビットで表される情報を私たち人間が認識する上では，それを1つの数値として扱い，私たちが普段使っている10進数ではなく16進数を用いて表現します．

そこで，この節では，まずコンピュータ・システムを理解する上で不可欠なビットの概念とその表現方法である2進数や16進数について学んでいくことにしましょう．

ビットとバイトの概念

図4-1で見たようにコンピュータ・システムでは，情報を伝えるための信号線が並列に10数本並べてつなげられています．信号線は電圧の高低というかたちで2つの状態を区別します．1本の信号線では最小の情報量である2つの状態しか表すことができませんから，信号線をいくつも並べることによって多くの状態を表すのです．信号線が2本なら，その各々が2つの状態を持つことができるので，(2×2＝) 4通りの状態を表すことができます．8本あれば（2の8乗＝）256通りの状態を表すことができます．このように信号線の数が増えるにしたがって，表現できる情報の量が増えることになります．

*コンピュータのソフト，ハードにわたるシステムの構造のこと．

信号線の数，つまり情報量を数えるための単位としてビットという言葉を用います．りんごを１個２個と数えるように，情報を１ビット２ビットと数えるのです．さらに８ビットの情報を１つの基本単位として，**１バイト**(Byte)と呼びます．メモリに記憶させたり，それを読み出したりする最小単位は１バイトであり，ビット単位で記憶や読み出しをすることはできません．

図4-2では，「ビット」と「バイト」がどのような関係にあるのかをメモリを例にして図示しておきました．この図に出てくるアドレス等の用語については，以降の「4.4節」のところでくわしく解説します．この図は，以降の節を読み進めた後でもう一度見直すと，よりいっそう理解が深まるでしょう．

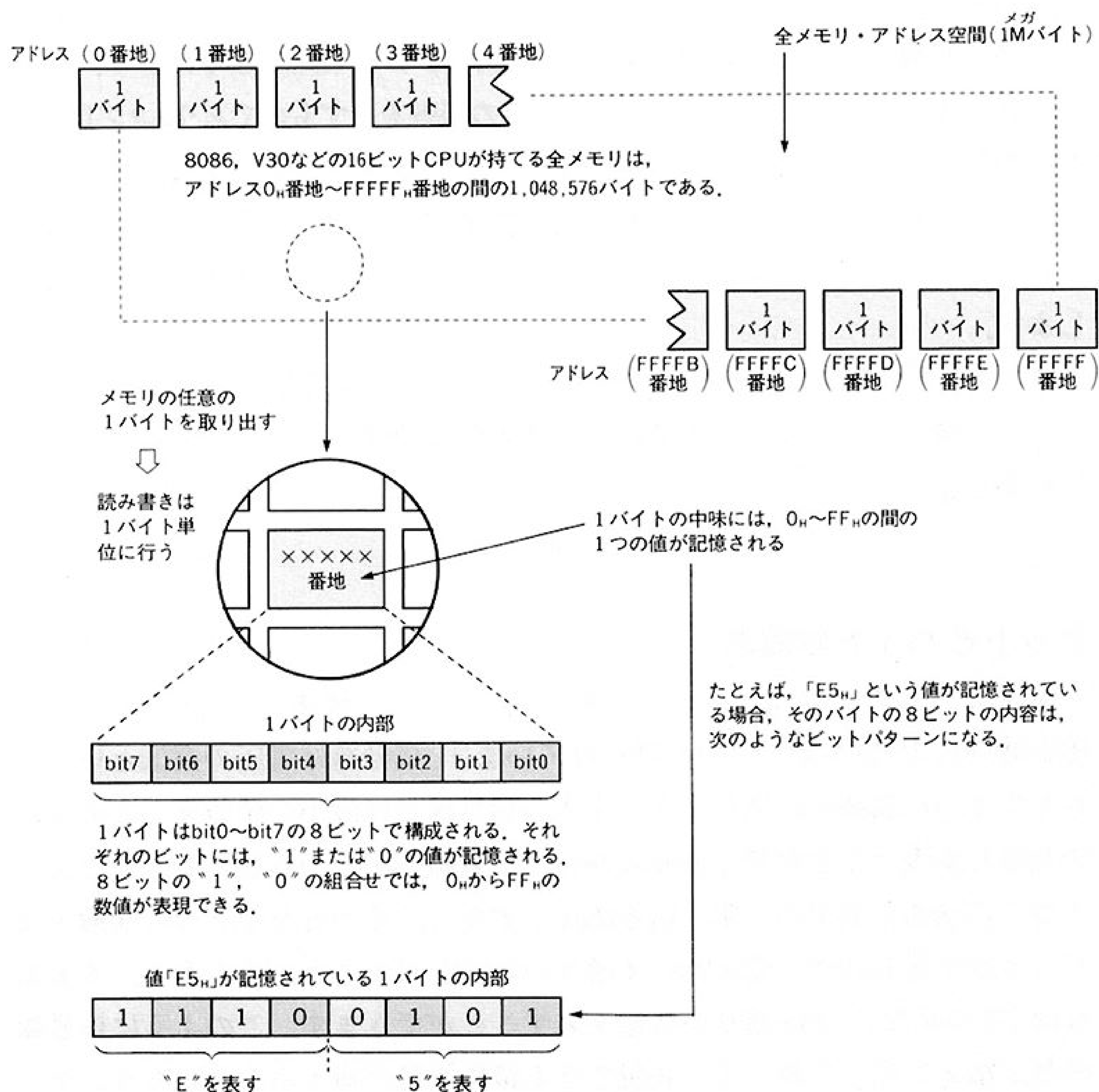


図4-2　ビットとバイト

ビットとバイトの表現方法　―2進数と16進数

1ビットの情報は電圧が高いか低いかという2つの状態のうちのどちらかを表しますが，これをそれぞれ「0」と「1」という数に対応させます．2ビットの組合せならば「00，01，10，11」の4つのうちのどれかの値になります．ビットの数が増えていけばいくほど，この組合せの数は増えていきます．1ビットについて「0」と「1」の2通りあるのですから，2をビット数だけ掛けた数の組合せがあるのです．8ビットならば2を8回掛けた256通りの組合せがあることになります．8ビット，つまり1バイトの中身は，次の図4-3のように図示します．

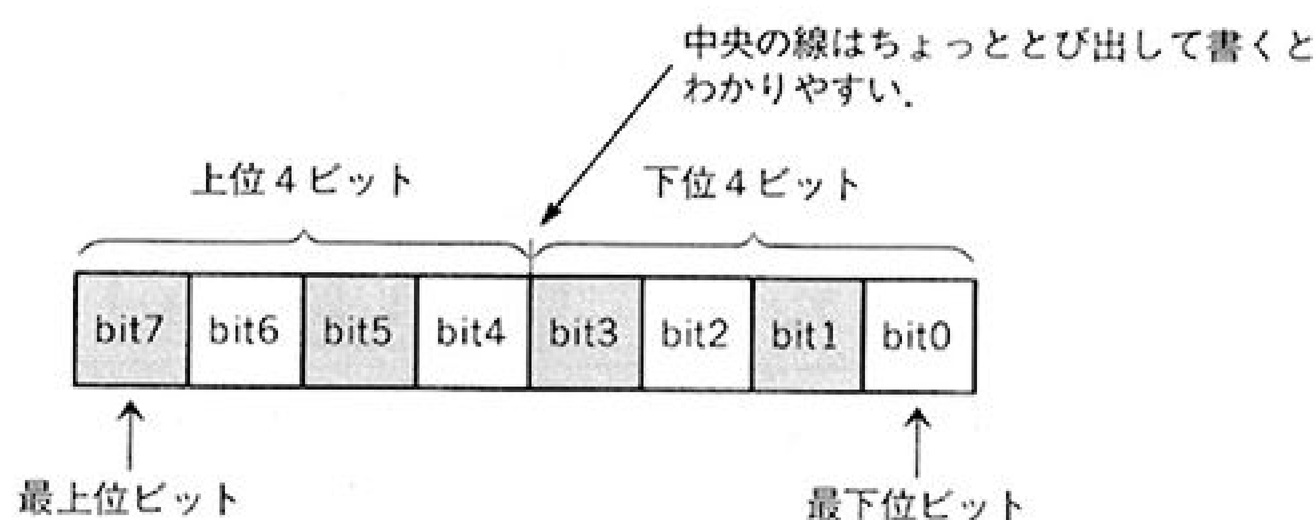


図4-3　1バイトのビットパターン

「0」と「1」で表す8ビットの組合せは「11100101」のように，2進数の数値と考えることができます．ただし2進数は桁数が大きくなると効率が悪いので，2進数で数値を表現することはあまりありません．通常このようなビットの組合せは**16進数**で表します．

私たちが普段から慣れ親しんでいる数値の表現方法は10進数ですが，コンピュータで扱う数値は16進数で表すのが普通です．すでに見てきたようにDEBUGで扱う数値もすべて16進数で表現しました．その理由は16進数による数値の表現方法がコンピュータの扱うデータにとって都合がよいからですが，くわしくは以後を読み進めていくうちに明らかになるでしょう．

10進数では1つの桁で0から9までの10種類の数値を表しますが，16進数では1つの桁で0から15までの16種類の数値を表します．そして10から15までの数値を1桁で表すために，AからFまでの英文字を数字として使います．また16進数であることを示すために後ろに「H」(Hexadecimal notation：16進表示）を付けます．

次の**表 4-1** に 16 進数の 1 桁で表せる数値を，10 進数，2 進数とも対応させた対照表の形で示しておきます．

10進	16進	2進の4ビット			
		bit^{3}_{7}	bit^{2}_{6}	bit^{1}_{5}	bit^{0}_{4}
0	0_H	0	0	0	0
1	1_H	0	0	0	1
2	2_H	0	0	1	0
3	3_H	0	0	1	1
4	4_H	0	1	0	0
5	5_H	0	1	0	1
6	6_H	0	1	1	0
7	7_H	0	1	1	1
8	8_H	1	0	0	0
9	9_H	1	0	0	1
10	A_H	1	0	1	0
11	B_H	1	0	1	1
12	C_H	1	1	0	0
13	D_H	1	1	0	1
14	E_H	1	1	1	0
15	F_H	1	1	1	1

表 4-1　10 進，16 進，2 進の対応表

16進数と2進数の関係を理解するために具体的な値で考えてみましょう．16進のC3$_H$のビットパターンは次の図4-4のようになります．

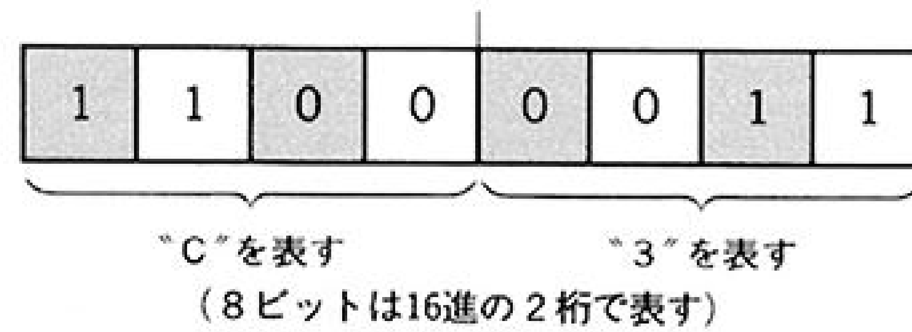


図4-4　16進のC3$_H$のビットパターン

この16進と2進の関係は，次のように考えると簡単に理解できます（図4-5）．

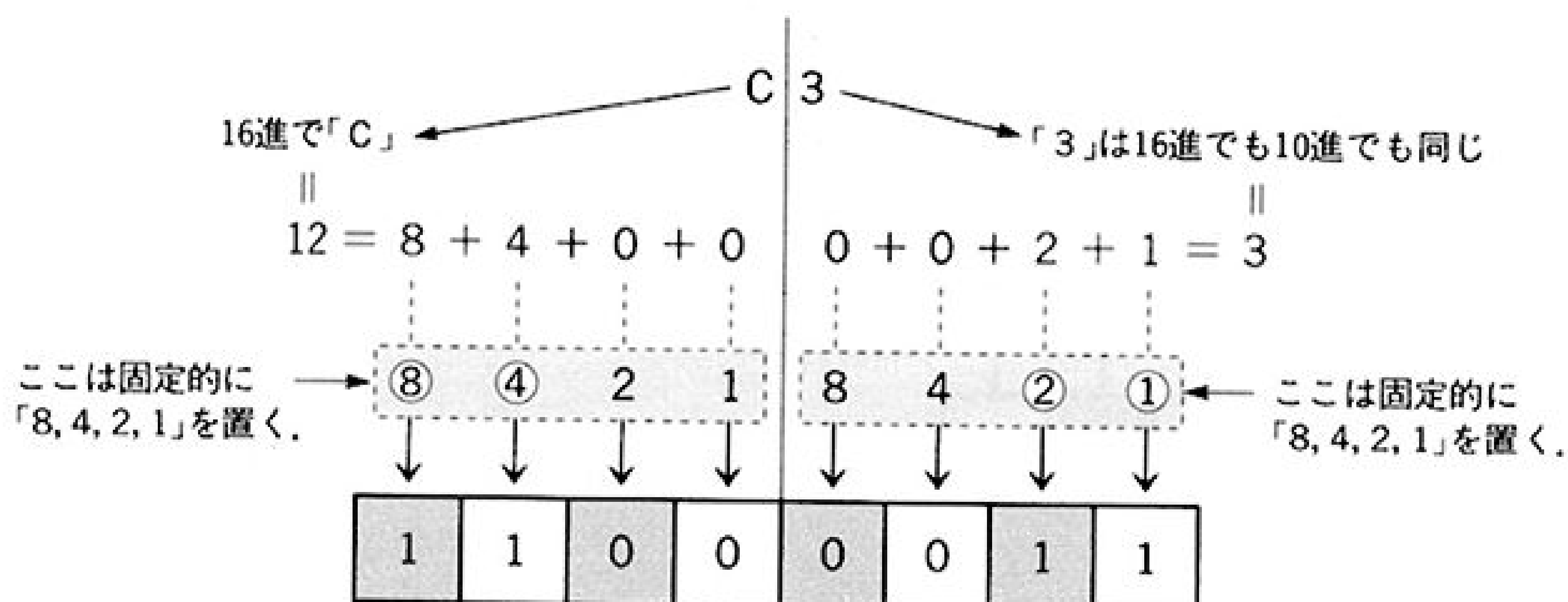


図4-5　16進→2進への変換

この図を見ているだけでも変換の方法がピンとくるかもしれません．ここは非常に重要なところなのでくわしく解説しておきましょう．

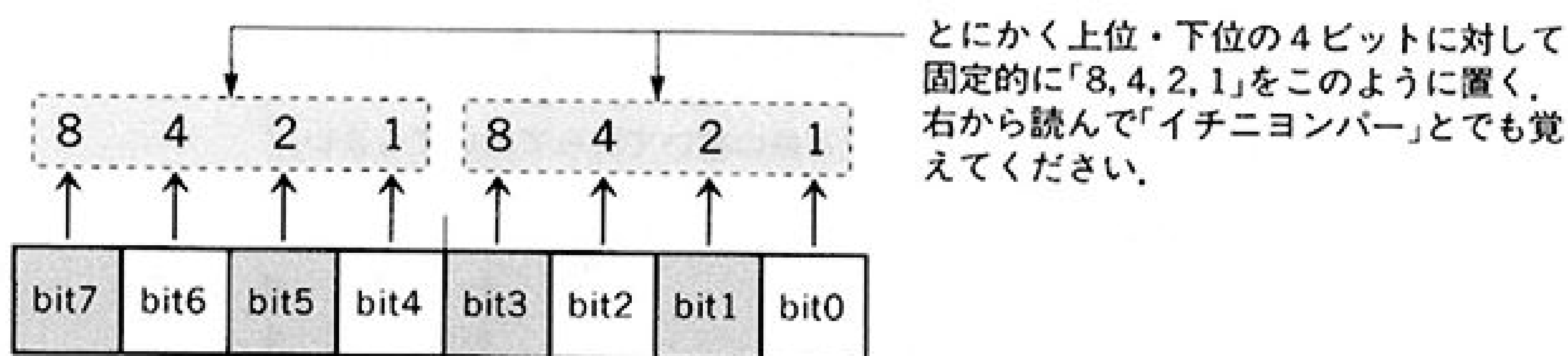


図4-6　2進と16進との関係

たとえば2進によるビットパターンがわかっている場合，1バイトの8ビットを上位4ビット／下位4ビットに分け，それぞれのビットに図4-6のように「8，4，2，1」の数値を固定的に持たせます．そしてビットパターンの「1」が立っているビットのところの数値だけを4ビットごとに合計します．その合計値が上位／下位4ビットのビットパターンをそれぞれ10進数で

表した値となります．10 進数から 2 進のビットパターンに直すには，この逆の操作を行えばよいわけです．

では，4 ビットのビットパターンから 10 進，16 進への変換の具体例を示します．たとえば，

10進	16進	2進の4ビット
10	A	1 0 1 0

図 4-7　10 のビットパターン

を例にとると，

$$8 + 0 + 2 + 0 = 10 = A_H$$

図 4-8　2 進→10 進→16 進への変換

となります．

ここで 10 進，16 進，2 進のビットパターンの変換方法の原理をまとめておきましょう（図 4-9）．まず，すべての数値を 16 進の 1 桁，つまり 2 進の 4 ビットに分けて考えます．そしてその 4 ビットを次のように変換します．

図 4-9　10 進，16 進，2 進ビットパターン間の変換方法

ではここで図4-4に戻ってみましょう．今度はもう図4-10を見るだけでわかると思います．

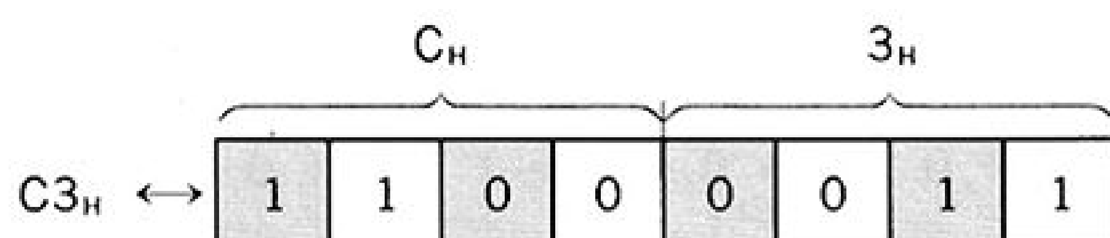


図4-10　$C3_H$の16進，2進変換

表4-1からわかるように，16進数の1桁で表せる「0_H」から「F_H」までの16種類の数値は4ビット（$2^4=16$）で表せるすべての数でもあります．したがってどんな16進数でも4桁ごとの2進数に変換することができ，逆に2進数も4桁ごとに区切ることによって16進数に容易に変換することができます．

ここまで理解すれば，2進数の世界を16進で表現すると都合がよいことは納得できたことでしょう．16進数を用いると2進の8ビット，つまり1バイトのデータを2桁でうまく表現できるのです．1バイトのビットパターンは，「00_H」から「FF_H」の16進数（10進数では，0から255まで）に対応し，256個の数を扱うことができます．

8086CPUは16ビットCPUであり，データとして16ビット（2バイト）の値を扱うことができます．8ビットをバイトと呼ぶのに対し，16ビットを**ワード**（WORD）*と呼んでいます．16ビットのデータは，8ビットの場合とまったく同様に考えて，16進の4桁で表します．したがって，16×16×16×16＝65536個の数を扱うことができます．

1ワードについて2進，16進の変換の具体例を最後に挙げておきます（図4-11）．1バイトの場合とまったく同じであることがわかるでしょう．単に桁数が増えたにすぎません．

＊ワードのビット幅はCPUの種類や慣例によって異なることがある．

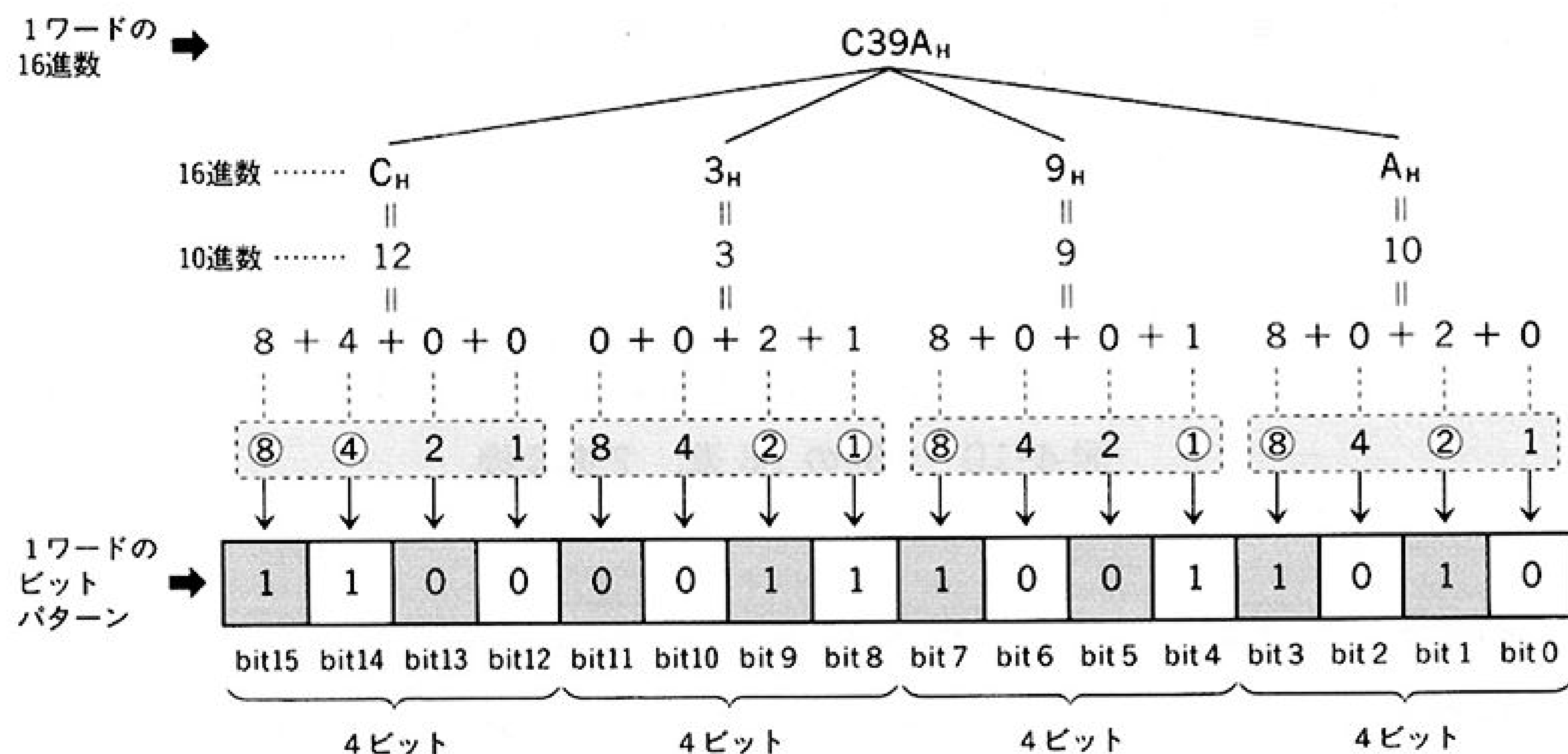


図4-11　1ワードの2進，16進変換

なお，1バイトおよび1ワードの10進－16進変換については，巻末のAPPENDIXで具体的な計算方法を解説しておきます．また，10進－16進－2進対応表もAPPENDIXに載せておきますので随時参照してください．

●

負の数

これまで解説してきたように8ビットでは，0から255までの256個の数値を表すことができます．実は，この256個の数値の半分を負の数と考えてもよいのです．具体的には「80_H」から「FF_H」までの128個の数値を－128から－1までの数値に割り当てます．残りを正の数に割り当てると，8ビットのビットパターン全体で－128から+127までの256個の数値を表せることになります．

これは次のように考えます．FF_Hに1を足すと100_Hとなりますが，8ビットは16進の2桁ですからオーバーフロー（桁あふれ）してしまいます．しかし，オーバーフローを無視すると00_Hとなり，FF_Hは0より1小さいと考えることができます．このようにして，FF_Hを－1とします．同様にFE_Hはさらに1小さいので－2，FD_Hは－3としていくのです．

−128, −127, −126, …　−3, −2, −1, 0,　1,　2, …,　126, 127

80_H,　81_H,　82_H, …　FD_H, FE_H, FF_H, 00_H, 01_H, 02_H, … , $7E_H$, $7F_H$

負の数を含む場合の10進，16進，2進の対応表を次の**表4-2**に示してみましょう．

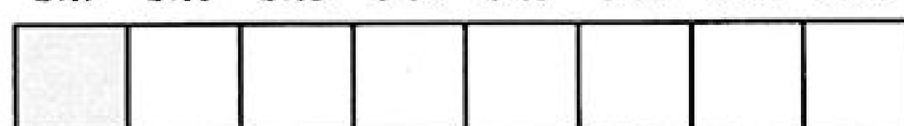
2進ビットパターン	16進	10進	2進ビットパターン	16進	10進
1 0000000	80_H	−128	0 0000000	00_H	0
1 0000001	81_H	−127	0 0000001	01_H	1
1 0000010	82_H	−126	0 0000010	02_H	2
⟆	⟆	⟆	⟆	⟆	⟆
1 1111101	FD_H	−3	0 1111101	$7D_H$	+125
1 1111110	FE_H	−2	0 1111110	$7E_H$	+126
1 1111111	FF_H	−1	0 1111111	$7F_H$	+127

負の数　　　　0または正の数

符号ビット



bit7　bit6　bit5　bit4　bit3　bit2　bit1　bit0



このビットが0ならば，0または正の数

このビットが1ならば，負の数

表4-2　符号付きの数の10進，16進，2進の対応表

「80_H」から「FF_H」までの負の数は，必ず最上位ビットが1になっていますね．逆に0または正の数では必ず最上位ビットが0になっています．つまり，最上位ビットが1ならば負の数であるということがわかるので，最上位ビットのことを**符号ビット**と呼びます．

このような負の数を用いるか用いないか（つまり16進数をどう見るか）は，プログラムによるデータの解釈しだいです．プログラマーが8ビットの値を符号付きであるとして扱うプログラムを書けばその値は符号付きになり，符号なしであるとして扱うプログラムを書けば符号なしになるのです．同じ値

を符号付きと考えるか符号なしと考えるかでどのように値が変わるかを次の図 4-12 に示します.

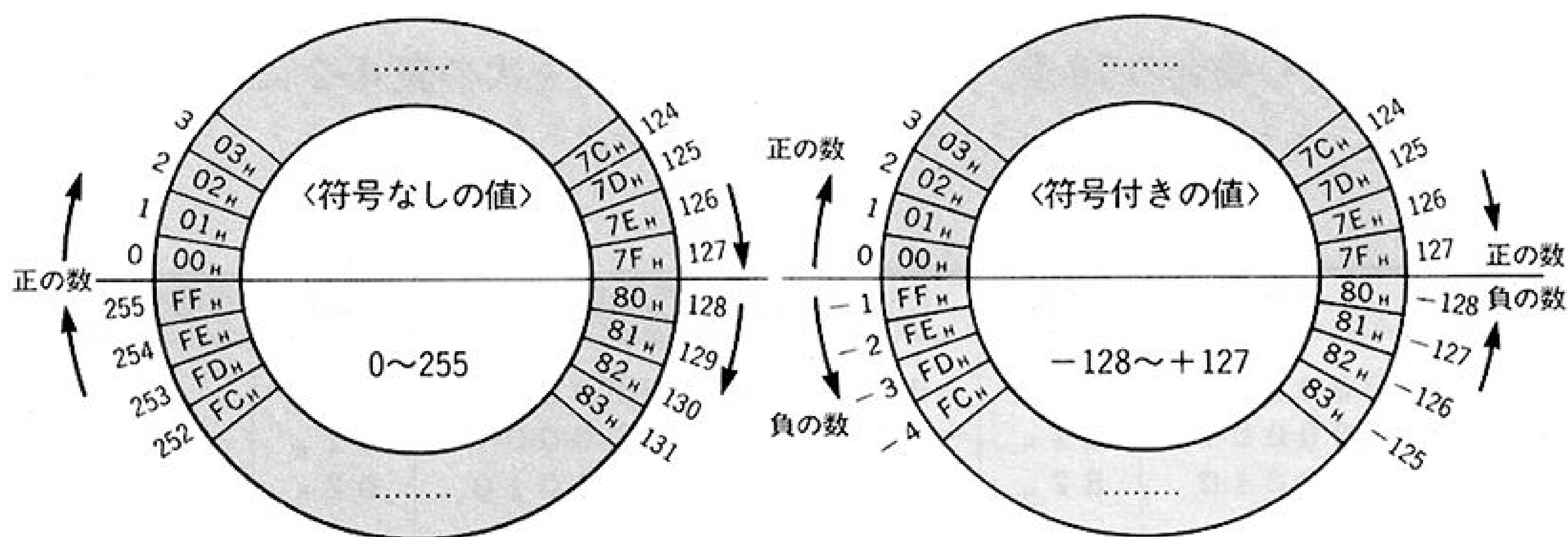
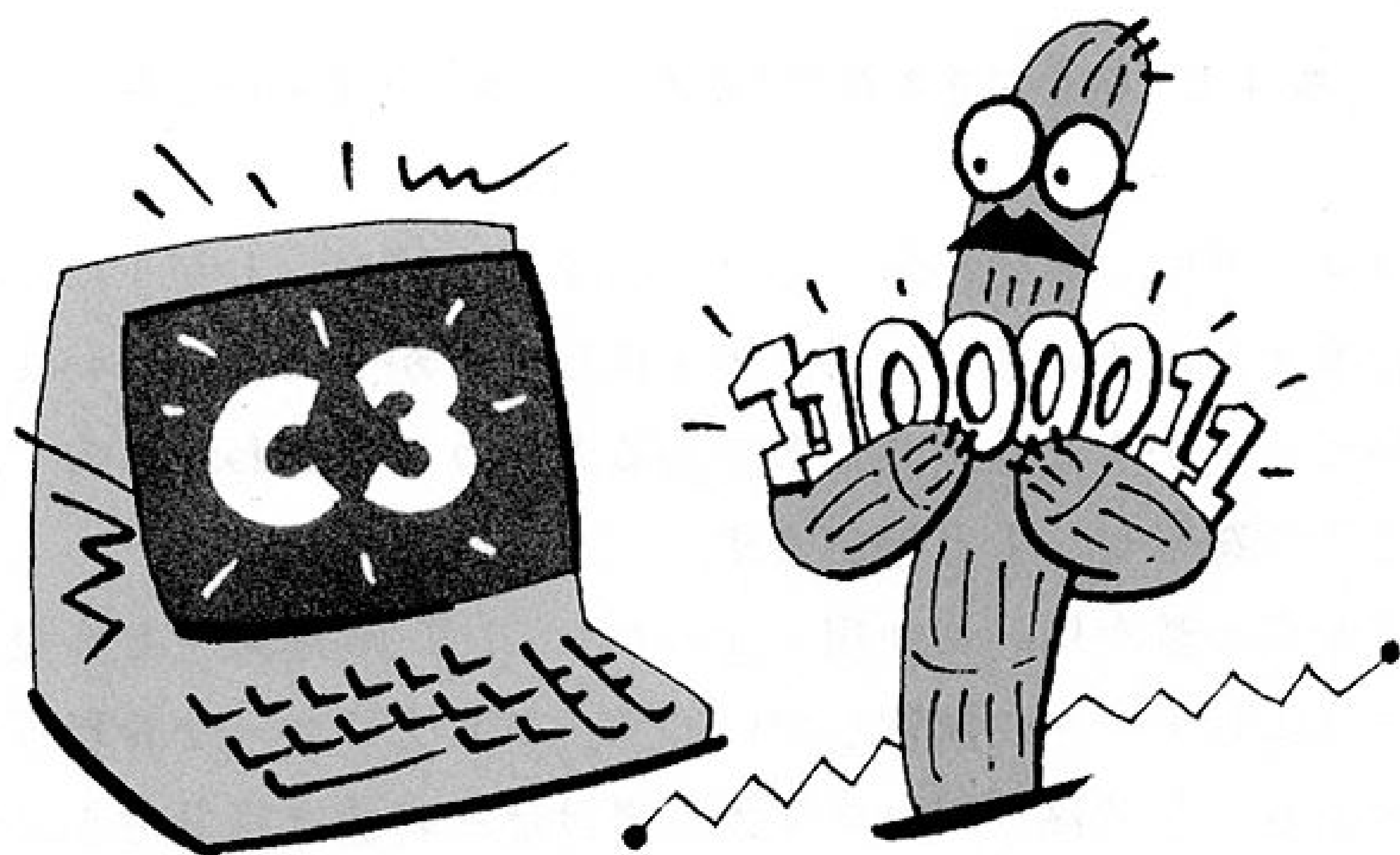


図 4-12　符号付きの値と符号なしの値

また，1 ワード，つまり 16 ビットで表せる 65536 個の数値についても同様に「8000_H」から「$FFFF_H$」までを−32768 から−1 までの負の数値と考えることにより，−32768 から+32767 までの数を表すこともできます．



4.3 CPU

CPUはコンピュータの頭脳にあたる部分です．CPUとは「Central Processing Unit」（中央処理装置）の略で，コンピュータ・システム全体を制御する働きを持っています．

CPUの内部にはデータを一時的に蓄えたり，計算したりするためのレジスタ群やプログラムの流れを管理するカウンタなどがあり，「マシン語の解読」，「データの読み書き」，「演算」，「判断」，「入出力」等の処理を行います．CPUはまさにコンピュータの働きの中心的存在なのです．

8086 CPUの具体的な各レジスタやその働きなど，くわしいことは5章で解説しますが，ここではその前にCPUの一般的な動作原理についてまとめておくことにします．

CPUの働き

CPUの基本的な動作は，メモリに格納されているマシン語命令を解読して，それを実行することです．この動作は，マシン語の1命令ごとに「命令の読み込み」，「命令の解読」，「命令の実行」という3つの過程に大きく分けられます．この3つの過程を完了することでマシン語の1命令の実行が完了します．1つの命令を実行するとすぐに次の命令の「読み込み→解読→実行」へと進み，またその次の命令の…というように，CPUはマシン語の命令を1つ1つ実行していくことによって，プログラム全体を実行します．

マシン語の命令は1バイトのものもあれば数バイトにわたるものもありますが，各命令にはそれぞれ異なるビットパターンが割り当てられています．CPUはこのビットパターンを解析して，その命令に対応する処理を実行するのです．命令の種類には，「演算」，「演算結果によるプログラム実行の制御」，「演算結果などのデータの転送」，「外部とのデータの入出力」などがあり，そ

のなかには CPU 内部だけで処理が行われるものと，メモリや I/O（入出力装置）とのデータのやりとりを伴うものがあります．

「データを転送する命令」を実行する過程や，マシン語命令の読み込みの過程などでは，CPU は決められたシーケンス（手順）にしたがってメモリや I/O とデータをやりとりします．CPU はアドレス，メモリと I/O を区別する信号，読み込みと書き込みを区別する信号などをタイミングをはかってバスに出力し，メモリや I/O はこの信号の指示にしたがってデータの入出力を行います．このようにメモリや I/O とやりとりするための指示を出すのも CPU の役割です．各信号やデータは，クロックという信号に呼吸を合わせることによって，互いにバラバラに動作することがないように，統制されたタイミングで出力されています．

次の図 **4-13** に，CPU の動作をまとめてみましょう．

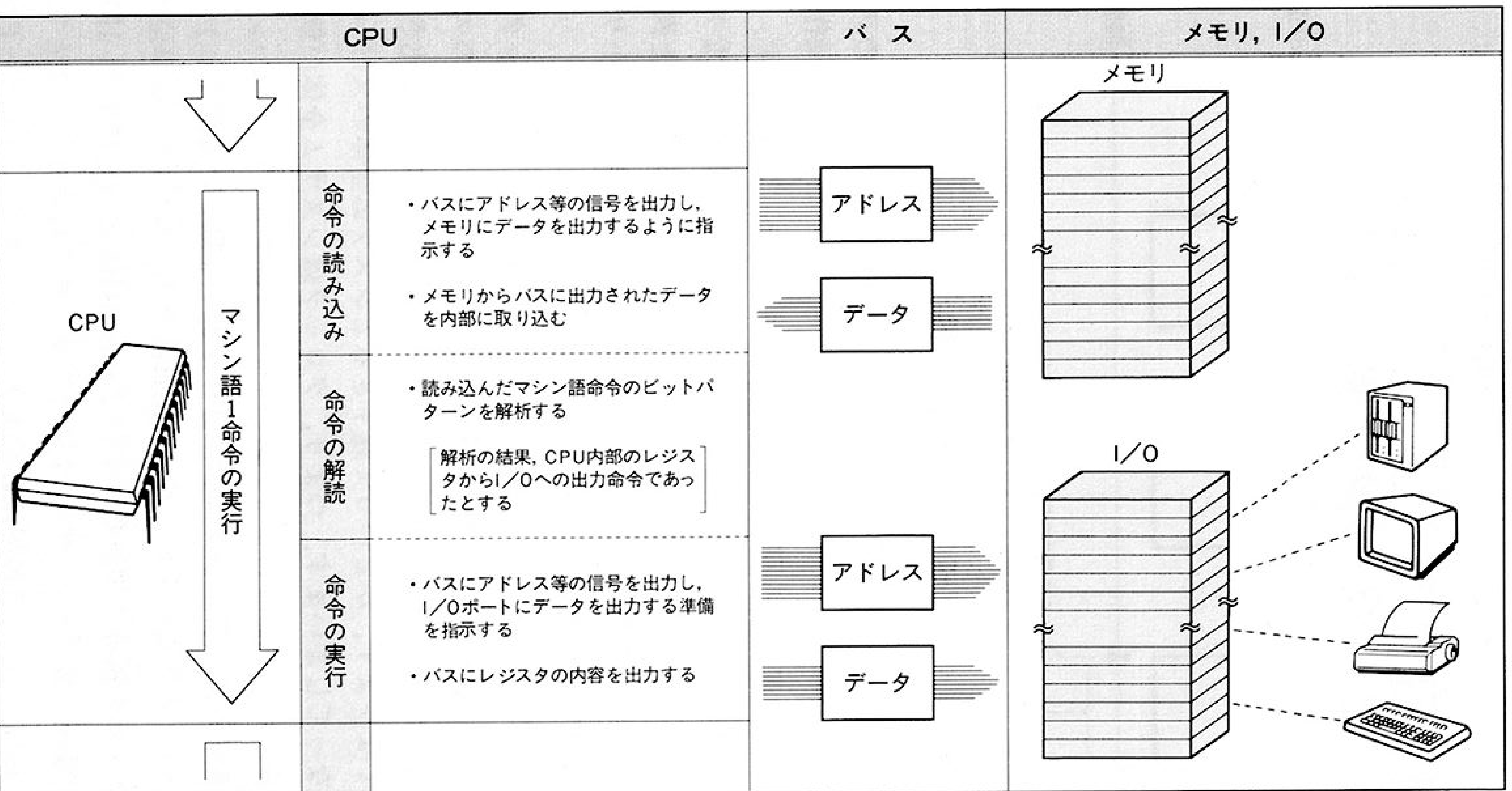


クロック

コンピュータの動作の電気的な仕組みについて解説しましょう．

コンピュータ・システム内の各時点における状態は，CPU，メモリ，I/O やそれをつなぐ信号線であるアドレスバス，データバス，その他の各種信号線の１本１本の電圧が高いか低いかで表され，さらにその状態によって次の状態が決定されます．コンピュータの動作は，このように電気的には電圧の状態の変化にすぎません．こういった電圧の状態の変化はてんでばらばらに起きるのではなく，クロックという同期信号によってシステム全体の動作のタイミングがとられています．

クロック（CLOCK）とは文字どおり時計の意味であり，クォーツ時計と同じように水晶発振器によって一定周期のパルスを出力しています．クロックの働きは非常に単純で，電圧の高い状態と低い状態を交互に作り出しているだけです．CPU はクロックの作り出す電圧の変化にタイミングを合わせて動作します．電圧が変化すると，しばらくは（人間の感覚からすると非常に短いが）その状態を保ちますから，その間にシステムの各部は次の状態に移る準備を整えます．そして，クロックの電圧が次の状態に変化した瞬間にタイミングを合わせて各部も次の状態へと変化します．



〈注意〉この図は概念図であり，実際の動作とは異なる．
実際には，命令の解読や実行の過程でバスが空いているときに，次の命令が読み込まれる．

図 4-13　CPU の動作

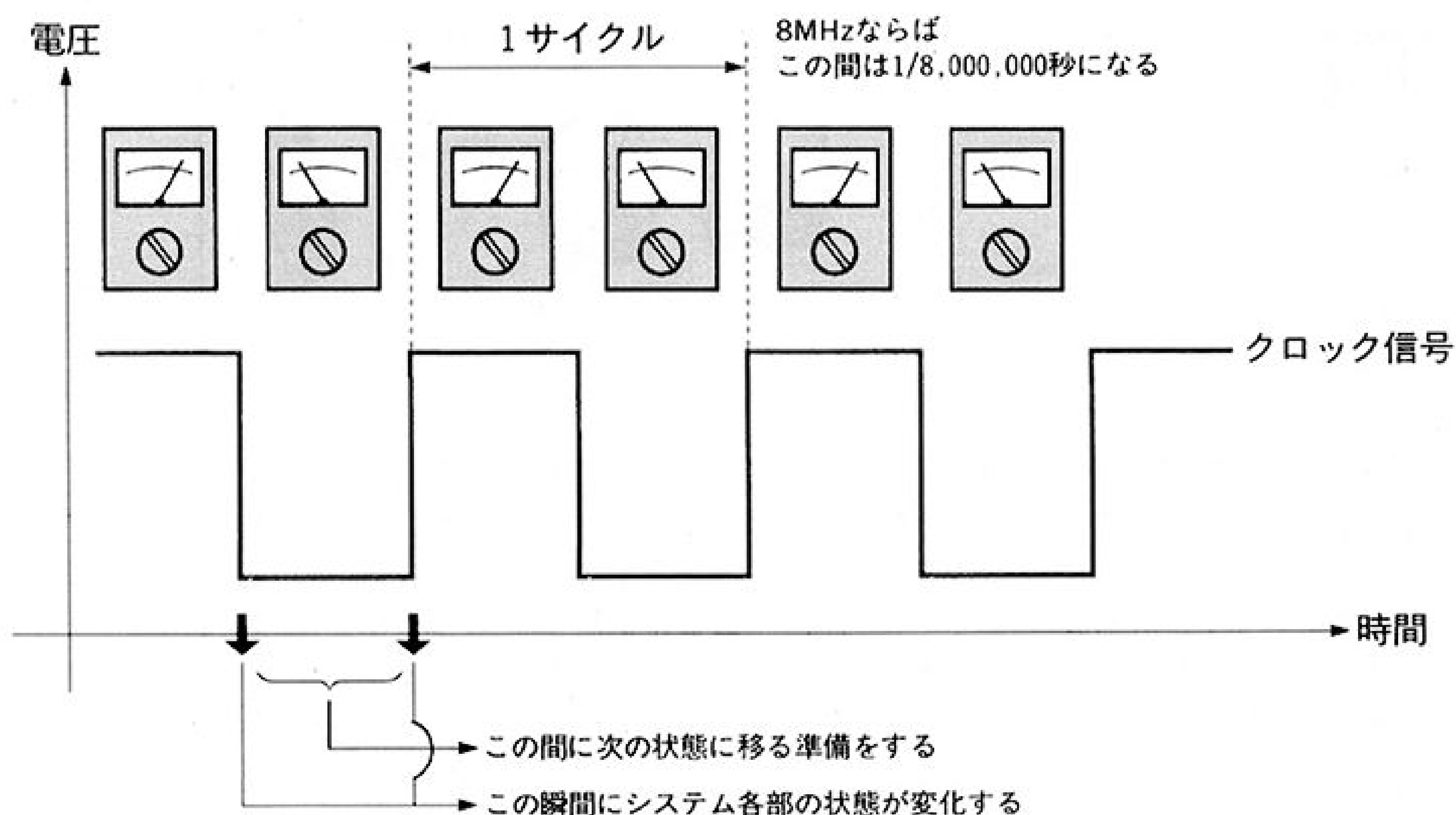


図 4-14　クロックの役割

CPU とメモリや I/O とのやりとりのタイミングは，ある一定回数のクロックの電圧の変化の回数によって決められています．このように，各動作が必要とするクロックの変化する回数のことを「クロック数」と呼びます．

CPU が実行する命令にはいろいろな種類がありますが，実行に必要なクロック数は命令の種類ごとに決まっており，それぞれの命令の実行速度を表しています．つまりクロック数が多ければその命令の実行には時間がかかり，少なければ短い時間で実行されるということです．

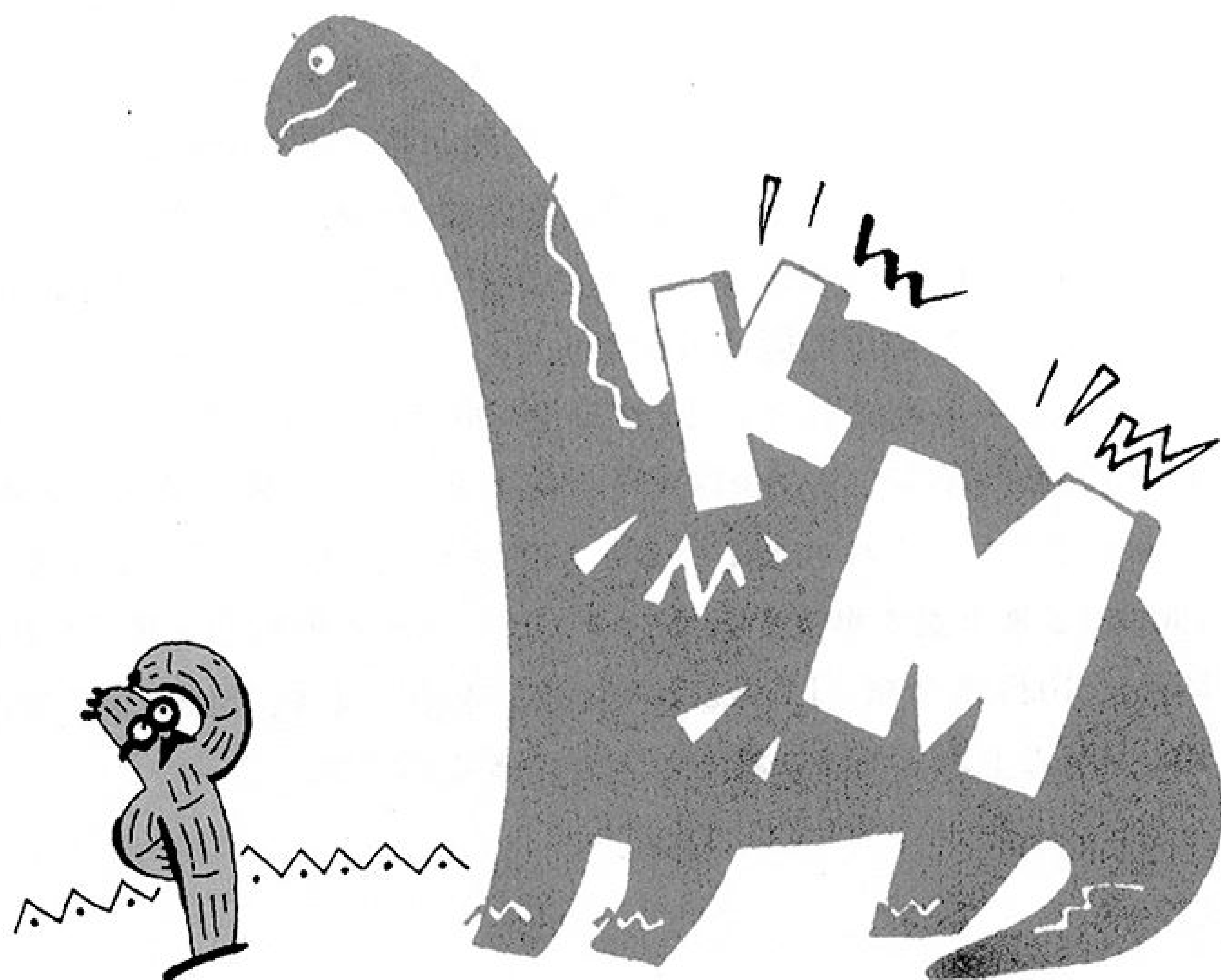
システムのすべての部分はクロックとともに動作するので，クロックのスピードが，システム全体のスピードを決定します．速すぎるとクロックのスピードに追いつかず，システムの各部で次のタイミングまでに状態を変化させることができなくなります．かといって遅すぎるとシステム全体のスピードがそのまま遅くなってしまいます．つまり，クロックはシステム全体が最も高速に動作しうるスピードに合わせてあるのです．CPU をコンピュータ・システムの頭脳とするならば，クロックは脈拍を作り出す心臓といえるでしょう．

コンピュータ・システムのクロックは，MHz（メガヘルツ）という単位で表します．FM ラジオの電波の周波数の単位である MHz と同じ単位で，1MHz は，クロックの電圧の変化が 1 秒間に 100 万回繰り返されるということです．CPU が同じならば，クロックの周波数が大きいほどシステムのスピードは速いということになります．

4.4 メモリ

メモリはプログラムやデータを記憶しておくところで，コンピュータにとって必要不可欠なものです．マシン語の解読，実行という過程を考えると，CPUの動作はそのほとんどがメモリとのやりとりであるともいえます．

メモリはバイト（8ビット）単位に区切られており，各バイトには0から始まる通し番号が付けられています．この番号をメモリの住所を表すものと考えてアドレス（番地）と呼びます．3章で取り上げたDEBUGのDコマンドでもこのアドレスを指定することにより，メモリの内容を表示しました．



アドレス　―K(キロ)とM(メガ)―

アドレスは，メモリを理解する上で非常に重要な概念です．このアドレスを指定することによって，CPUは何十万もあるメモリの中から目的の1つのメモリを選択し，データをやりとりすることができるのです．8086CPUでは，アドレスは0_H番地から始まり（1_H番地からではない！），$FFFFF_H$番地まで存在します．このことを「8086CPUのメモリ空間は00000_H番地から$FFFFF_H$番地までの1Mバイトである」というような表現をします．また，アドレスは0_H番地に近い方を「低い」，$FFFFF_H$番地に近い方を「高い」という言い方をします．

メモリ容量を表す場合には256Kバイトとか，384Kバイトなどといいますが，これはそのコンピュータに1バイトのデータを256K個，あるいは384K個格納できるだけのメモリが実装されているという意味です．K（キロ）は「kg」とか「km」のK（キロ）と同じ意味ですが，コンピュータで扱う数値の場合，1000ではなく1024を意味します．なぜ1024なのかというと，1024は2の10乗，16進数で書くと400_Hという切りのいい数字になるからです．つまり256Kバイトといっても256,000バイトではなく，実際には262,144（256×1024）バイトもあるわけです．コンピュータは2進数の世界ですから，2の累乗を単位としたほうがすっきりした数字で表すことができます．

256Kバイト実装という場合は，0_H番地からメモリが実装されているとすると$3FFFF_H$番地までメモリが実際に取り付けられているということです．384Kバイトなら$5FFFF_H$番地までです．メモリ容量とアドレスの関係をわかりやすく表したのが以下の図4-15です．

この表からわかるように16進で10000_Hバイトが64Kバイト，20000_Hバイトが128Kバイトです．8086CPUが扱えるメモリの最大量は100000_H（1024K）バイトですが，これを1Mバイトと呼びます．M（メガ）は100万，つまり1000の2乗を表す単位であり，コンピュータの世界ではK（キロ）の場合と同様に1024の2乗（1,048,576バイト）を表します．8086CPUでは，なんと100万個以上のメモリを扱うことができるのです．

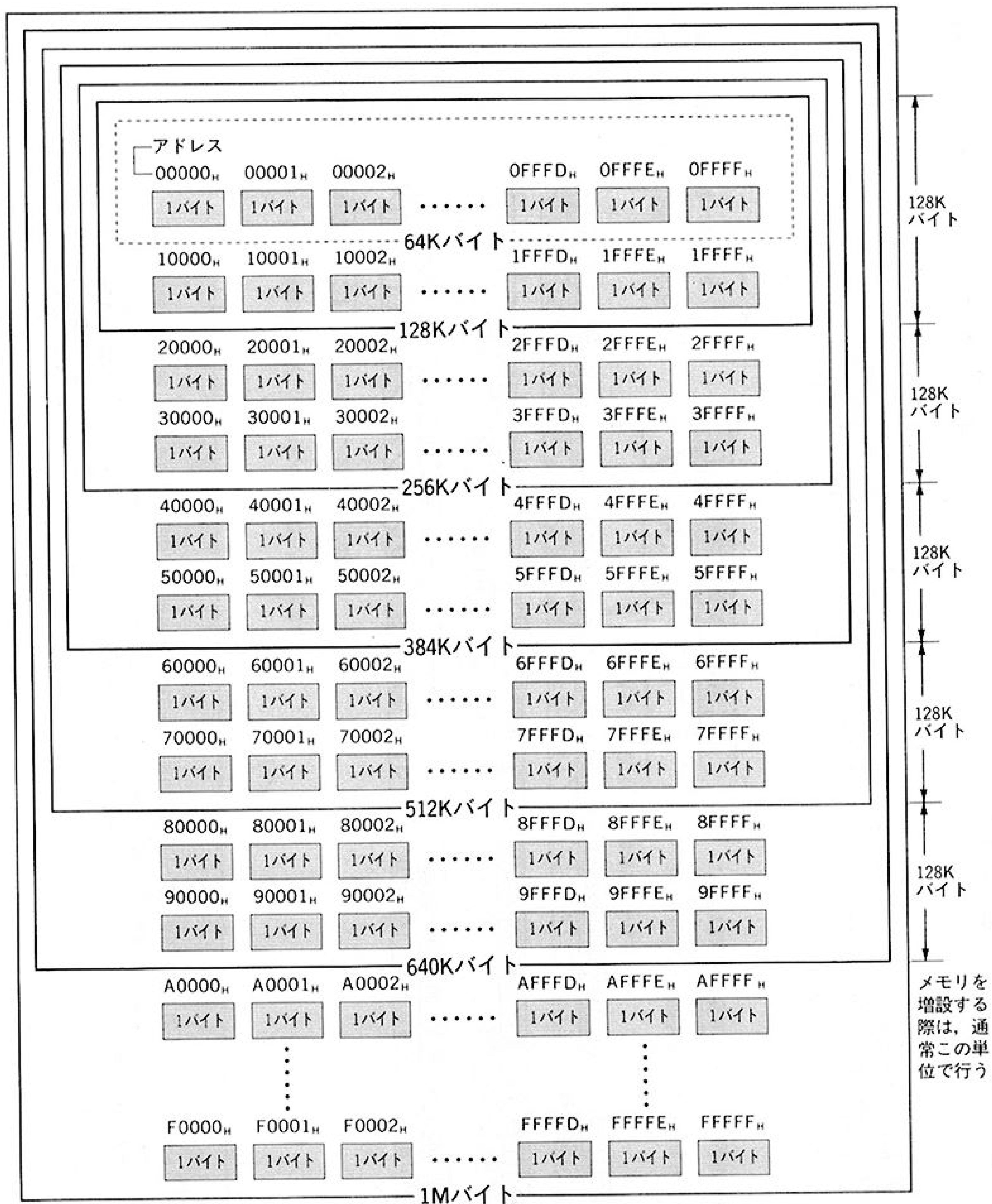
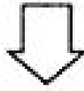
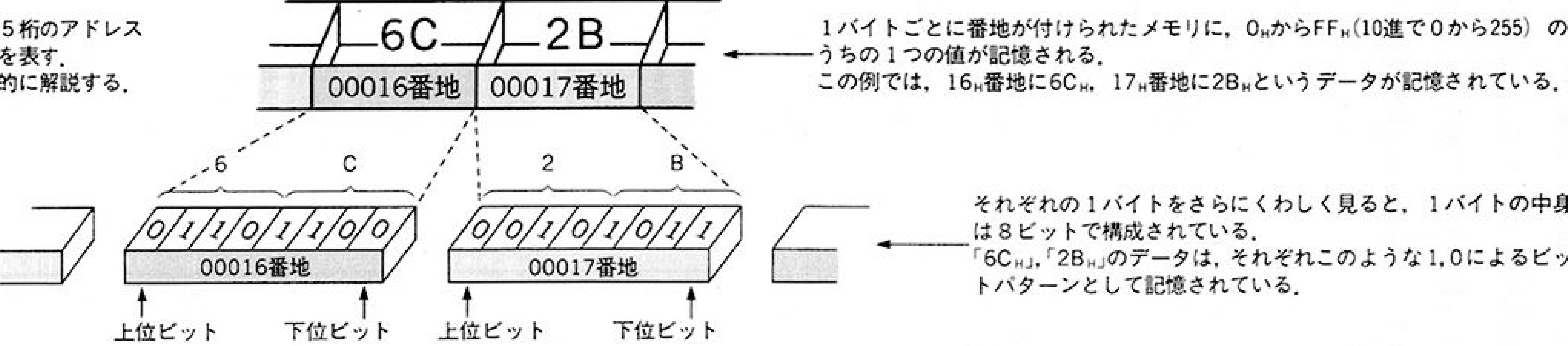


図 4-15　メモリとアドレス

メモリは1つが8ビットの大きさを持ち，0_Hから始まる番号の付いた箱を並べたものと考えることができます．以下の図 4-16 では，DEBUG の D コマンドを使ってアドレス 0_H番地からメモリの内容を表示させたときの実行例をもとに，「アドレス」，「バイト」，「ビット」の関係をわかりやすく図にしてみました．

```
A>DEBUG
-D 0000:0000 002F
0000:0000   33 13 AB 28 36 09 80 FD-F0 08 80 FD 36 09 80 FD   3.+(6..}p..}6..}
0000:0010   36 09 80 FD DD 0A 6C 2B-F8 50 60 00 4D 59 4B 2C   6..}].l+xP`.MYK,
0000:0020   6C 05 6C 2B 44 0E 80 FD-37 09 80 FD 37 09 80 FD   l.l+D..}7..}7..}
-
```





0000:0000	33	13	AB	28	36	09	80	FD	F0	08	80	FD	36	09	80	FD
	00000	00001	00002	00003	00004	00005	00006	00007	00008	00009	0000A	0000B	0000C	0000D	0000E	0000F
0000:0010	36	09	80	FD	DD	0A	6C	2B	F8	50	60	00	4D	59	4B	2C
	00010	00011	00012	00013	00014	00015	00016	00017	00018	00019	0001A	0001B	0001C	0001D	0001E	0001F
0000:0020	6C	65	6C	2B	44	0E	80	FD	37	09	80	FD	37	09	80	FD
	00020	00021	00022	00023	00024	00025	00026	00027	00028	00029	0002A	0002B	0002C	0002D	0002E	0002F

〈注意〉メモリの内容は，使用している機種によって異なる。

図4-16 メモリ（アドレス，バイト，ビット）の概念図

CPUとメモリ

CPUにはアドレスバスと呼ばれる20本の信号線と，データバスと呼ばれる16本の信号線が接続されています．CPUはメモリからデータを読み出したり，書き込んだりするときには，まずアドレスバスにアドレスデータを出力し，1Mバイトのうちのただ1つの番地を指定してメモリとの間で読み書きを行います．CPUがメモリを読み出すことをメモリのリード（Read），書き込むことをメモリへのライト（Write）とも言います．

データバスは16本の信号線からなるので，一度に16ビットのデータをCPUとメモリの間でやりとりすることができます．つまり2バイトのデータを一度に読み書きできるのです．また，この16本のうちの8本だけを使って1バイトのデータを読み書きすることもできます．

これらの様子を次に図解しておきましょう（図4-17）．

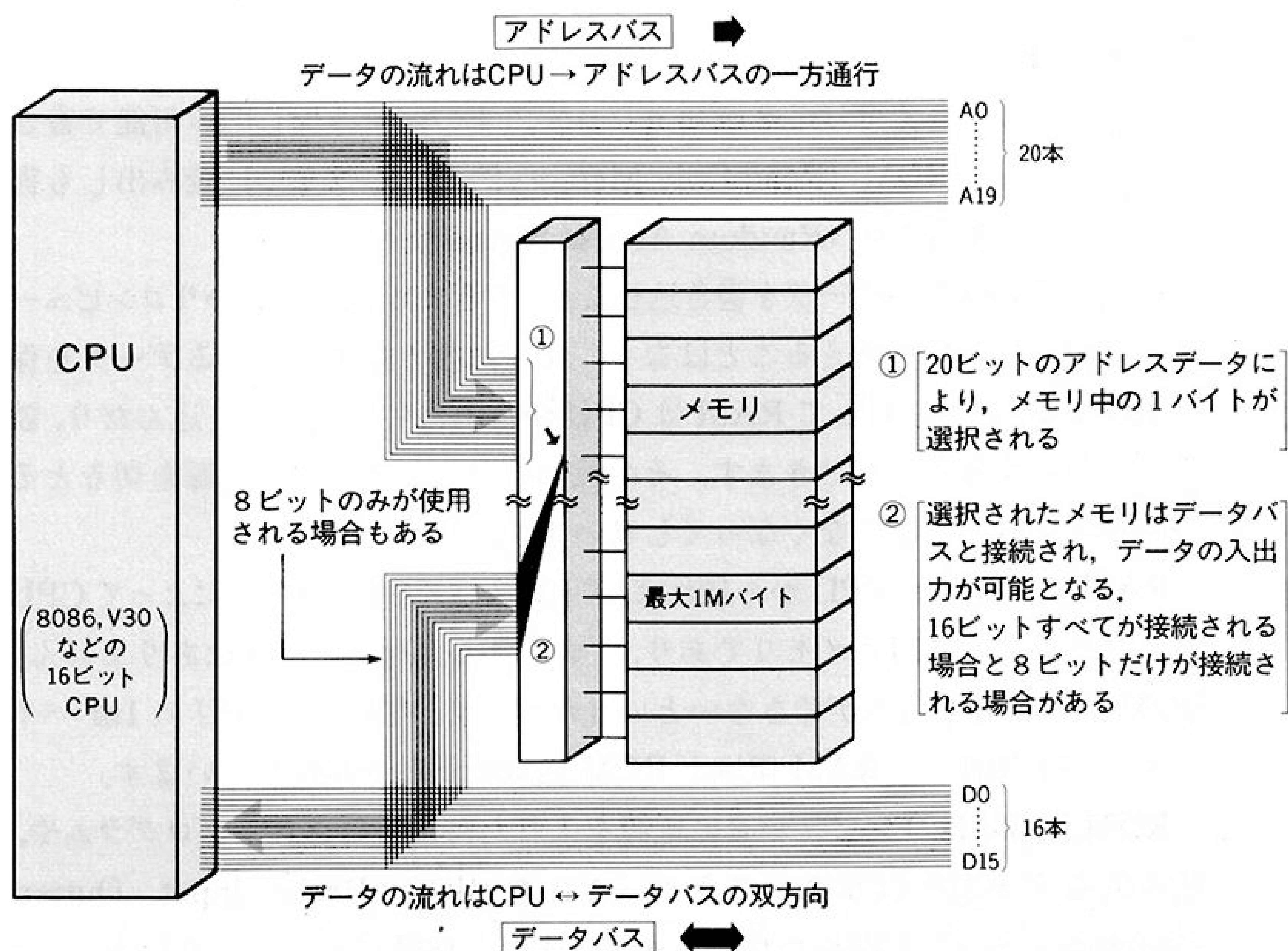


図4-17　メモリのリード／ライト

Z80などの8ビットCPUでは，最大64Kバイトまでのメモリしか扱うことができませんでした．これはCPUがメモリのアドレスを指定するアドレスバスが16本しかなかったからです．これに対し，8086CPUでは20本のアドレスバスがあるので最大1Mバイトのメモリを扱うことができます．

また，8ビットCPUにはデータバスに8本の信号線しかありません．したがってCPUからメモリへのリード／ライトは必ず8ビット，すなわち1バイト単位で行われます．

CPUを8ビット，16ビットに分類する基準は，このデータバスの信号線の本数です．データバスが16本ある16ビットCPUは，8本しかない8ビットCPUに比べて2倍のデータを一度にメモリとやりとりできるので，高速な実行が可能なのです．



RAMとROM

メモリには大きく分けて2種類あります．1つは読み出しのみ可能で書き込みのできないROM（Read Only Memory）* で，もう1つは読み出しも書き込みもできるRAM（Random Access Memory）です．

ROMにはCPUがデータを書き込むことはできません．その代わりコンピュータの電源を切っても消えることはなく，すでに書き込まれているデータを保ち続けます．これに対してRAMはCPUが自由にデータを書き込んだり，読み出したりすることができます．その代わりコンピュータの電源を切るとその内容はすべて消えてなくなってしまいます．

RAMもROMもCPUから見れば，アドレスバスとデータバスによってCPUにつながっている同じメモリであり，プログラム実行上の区別はありません．ROMはただ書き込みができないというだけです．実際8086CPUの1Mバイトのメモリ空間にはRAM領域とROM領域の両者が混在しています．

ROM領域にはコンピュータに電源を入れた時に実行されるプログラムや，基本的な入出力を行うプログラム（これをBIOS（バイオス）＜Basic Input Output System＞と言う）が納められています．また，機種によってはBASICイン

---

* ROMもランダムアクセスができるのでRAMなのであるが，慣用的に読み書きともに可能なメモリをRAMということになっている．

タープリタのプログラムが含まれているものもあります．ROMには消えてしまっては困るもの，書き換える必要のないものが入っているのです．

一方，ユーザープログラムや各種データエリア，プログラム実行のために必要な値を置いておくための作業領域（ワークエリア）などは，自由に書き込みや読み出しができなければならないので，RAM領域に置かれます．

8086CPUは電源ON時やリセット時には，アドレスの上位$FFFF0_H$番地からプログラムの実行が開始されるので，この内容は定まっている必要があります．また，6章で実習する割り込み処理のプログラムのアドレスを格納しておくための「割り込みベクタテーブル」というものが，アドレスの最下位0_Hから$3FF_H$までに割り当てられています．割り込みベクタテーブルは読み書きともに可能でなければなりません．

このため高位アドレス（$FFFFF_H$番地に近い方）をROM領域，低位アドレス（0_Hに近い方）をRAM領域というシステムが一般的です．MS-DOSもこのようなメモリ割り当てを仮定して設計されています．以下の図4-18にMS-DOSマシンの一般的なメモリ構成を図示しておきます．

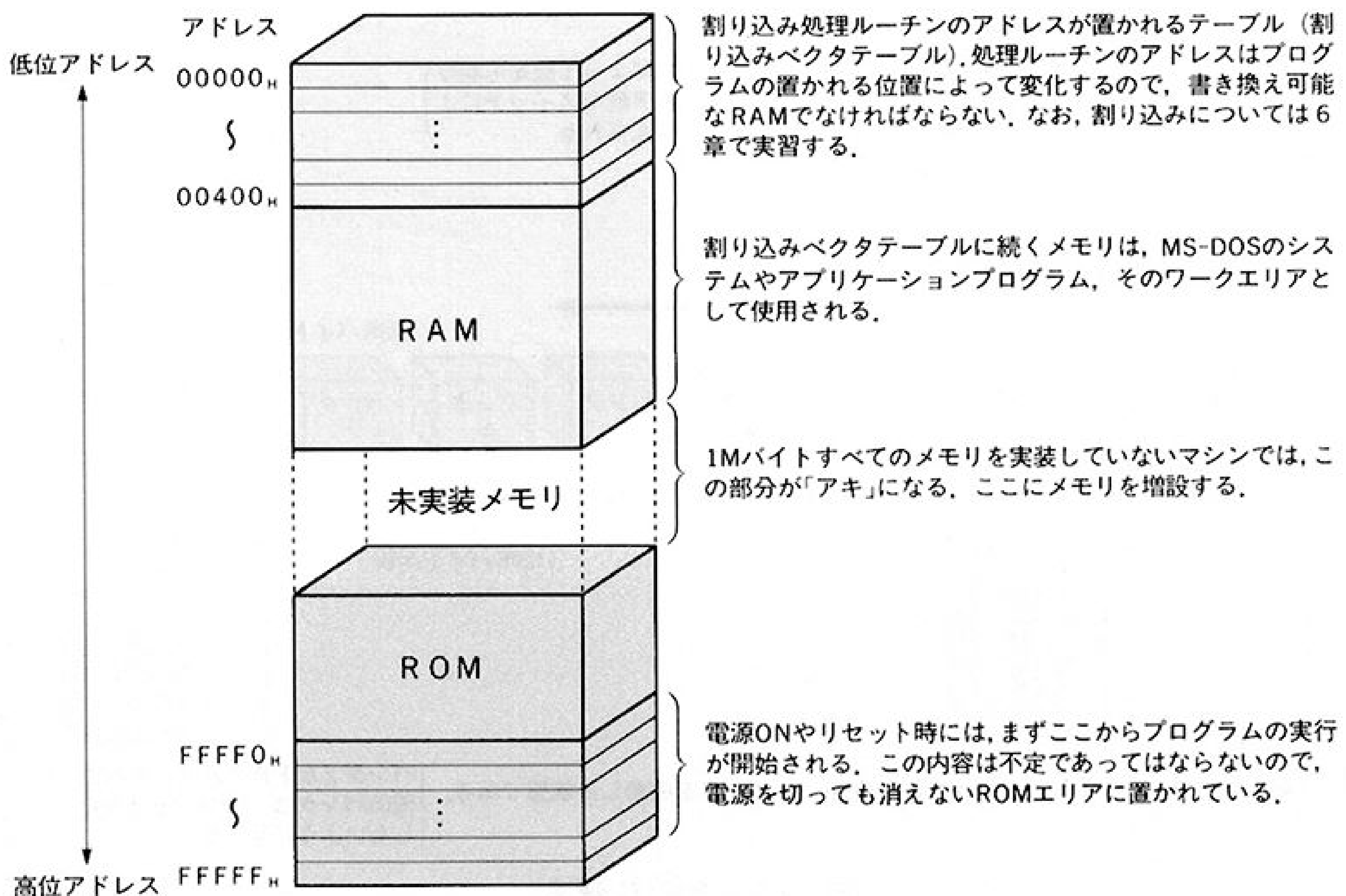


図4-18　MS-DOSマシンの一般的なメモリ構成

COLUMN

バンク切り替え　―RAMディスクの仕組み―

最近，メモリの大容量化，低価格化にともない，メモリの一部をディスクとして使用する「RAMディスク」が使われることが多くなってきました．これは，フロッピーディスクでは時間のかかった各種の処理を高速化することが目的です．かな漢字変換用の辞書などを入れるため，RAMディスクはさらに大きな容量を要求されるようになり，メインメモリの一部を割り当てるだけでは不足してしまい，必要な容量を満足するには，8086CPUが扱える上限である1Mバイトのメモリ空間に収まらなくなってきました．

1Mバイトというメモリ空間を超えるためのテクニックの1つがバンク切り替えという方式です．

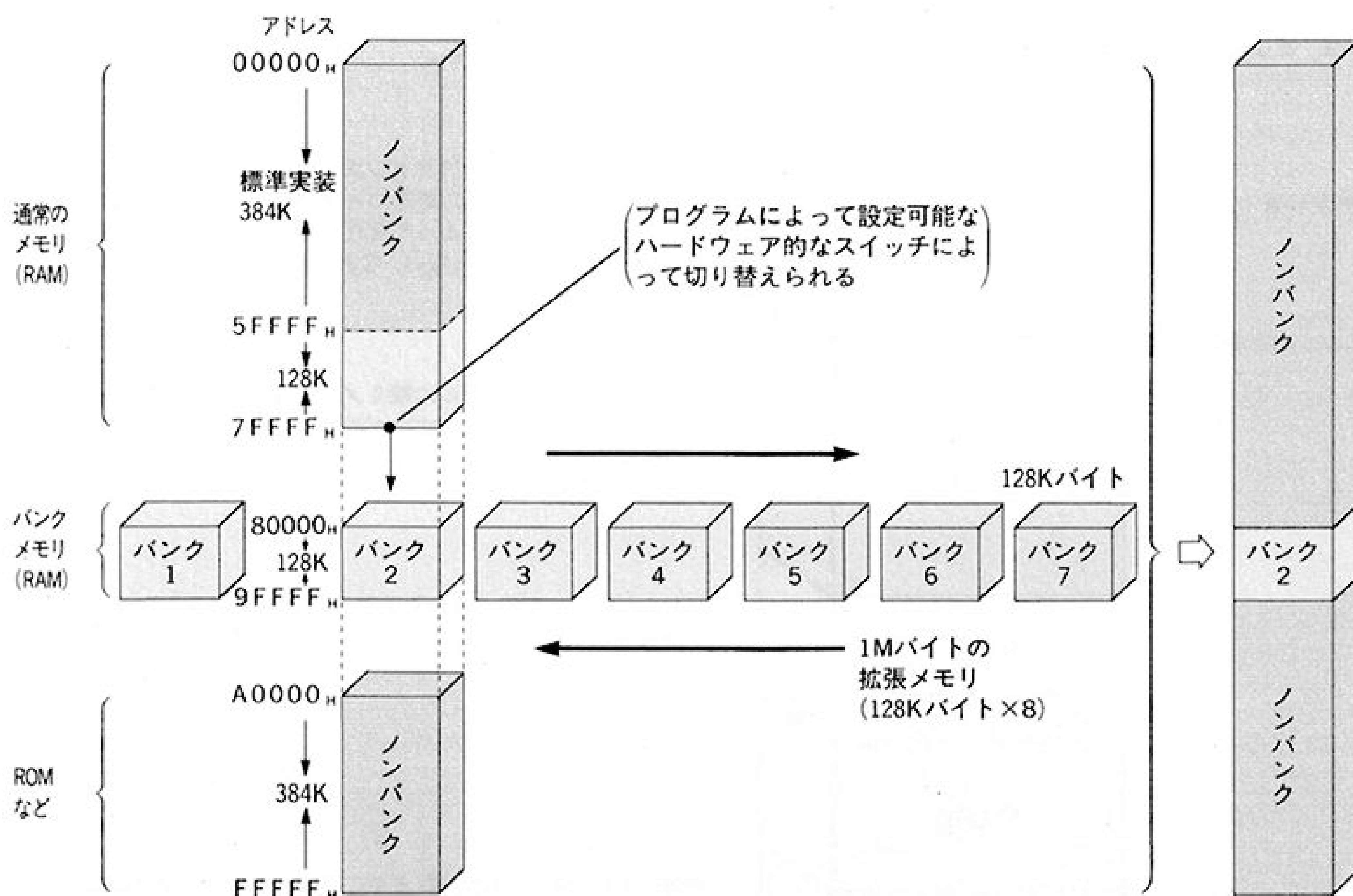


図　バンク切り替え

この図ではアドレス80000_Hから$9FFFF_H$までの128Kバイトのメモリ領域に7個のバンクメモリが存在しています．ハードウェア的なスイッチによって，これらのうちのどれか1つを選択し，アドレスバス，データバスにつながったメモリとして使用することができます．CPUからは，同一のアドレスにあるバンクメモリのうち，ただ1つがメモリとして見えており，他のバンクメモリは存在しないように見えます．全メモリ容量が1Mバイト以上であるとしても，ある時点でCPUにアドレスバス，データバスを介してつながっているメモリは最大1Mバイトであることには変わりないのです．このようなCPUから見えていないバンクを「裏バンク」と呼ぶこともあります．

各バンクを切り替えるには，バンクを切り替えるハードウェアに，ある信号を送ることで行います．プログラムによってその信号を送出すると，ハードウェア的なスイッチが切り替わって目的のバンクが選択されるという仕組みです．

MS-DOSシステムはバンク切り替え方式のメモリ管理を想定していないので，一般のプログラムでこの裏バンクまでをメモリ領域として使用することはできません．しかし，RAMディスクドライバ(RAMディスク用のプログラム)をシステムに組み込むと，RAMディスクとして使用することができます．RAMディスクはMS-DOSシステムから見ると通常のディスク装置と同じように見えます．RAMディスクドライバではシステムから転送命令を受け取ると，バンクを裏バンクに切り替え，メモリの内容を転送して，またもとのバンクに戻すという処理を行います．

また，バンク切り替えは，多くの8ビットパーソナルコンピュータで，64Kバイトというメモリ空間の上限を超えるため，あるいは同じアドレスのメモリ領域にROMとRAMを割り当てるために使われているテクニックでもあります．

4.5 I/O

「I/O」とは Input/Output System, つまり入出力装置のことです. コンピュータは, CPU とメモリの間だけでデータを処理していたのでは何の役にも立ちません. キーボードからの入力, CRT やプリンタへの出力, ディスクなどとのデータのやり取り, 他のコンピュータと回線を通じた通信などを行うことで, 初めて意味があるのです.「CPU に外部からデータを与え, 処理した結果を外部に伝える」という入力および出力ができるからこそコンピュータの行う処理に価値があるのです.

また, 最近クーラーなどの家庭電化製品にも組み込まれているコンピュータ（組み込みのコントローラ）では, 機械からいろいろな状態信号（ステータス信号という）を受け取り, それに応じて機械をコントロールするためのさまざまな信号を出力しなければなりません.

こういった信号のやりとり, つまり入出力するということは, 具体的にはデータバスに周辺装置の信号線を接続するということです. その接点にあたる部分が I/O ポート, すなわち入出力ポートです. ポート（Port）は「港」を意味しますが, その名の通りポートを介して「データ」や, データのやり取りをスムーズに行うための「ステータス信号」などを受渡し（入出力）するのです.

8086CPU では, ポートの数は通常最大で 65536 個まで設けることが可能です. CPU とのやりとりは, 20 本のアドレスバスの中の 16 本の信号線で表せる 0000_H ～ $FFFF_H$ のポートアドレスによって必要なポートを指定し, データバスによってデータをやりとりします.

I/O 選択のメカニズムは前節で解説したメモリの選択と同様です. メモリの場合はアドレスバスの 20 本の信号線をフルに使って 00000_H ～ $FFFFF_H$ の 1M バイトのメモリ空間のうちの 1 つを選択しますが, I/O ポートの場合はアドレスバスの下位 16 本の信号線で接続されている 0000_H ～ $FFFF_H$ の 64K 個(65536

個）のポートのうちの１つを選択します．

メモリをアクセスする場合もI/Oポートをアクセスする場合も同じアドレスバス，データバスを使用しますが，どちらをアクセスするかを区別するための信号がアドレスとともに出力され，その信号によってハードウェア的にメモリとI/Oを切り替えます．8086CPU以外のCPU* のなかにはこの信号が存在しないものがあります．そのようなCPUでは一部の特定アドレスにI/Oポートが接続されており，メモリとI/Oをアドレスのみで区別しています．

I/Oポートによるポートの選択と，入出力データの関係などを最後にまとめとして図示しておきましょう（図4-19）．

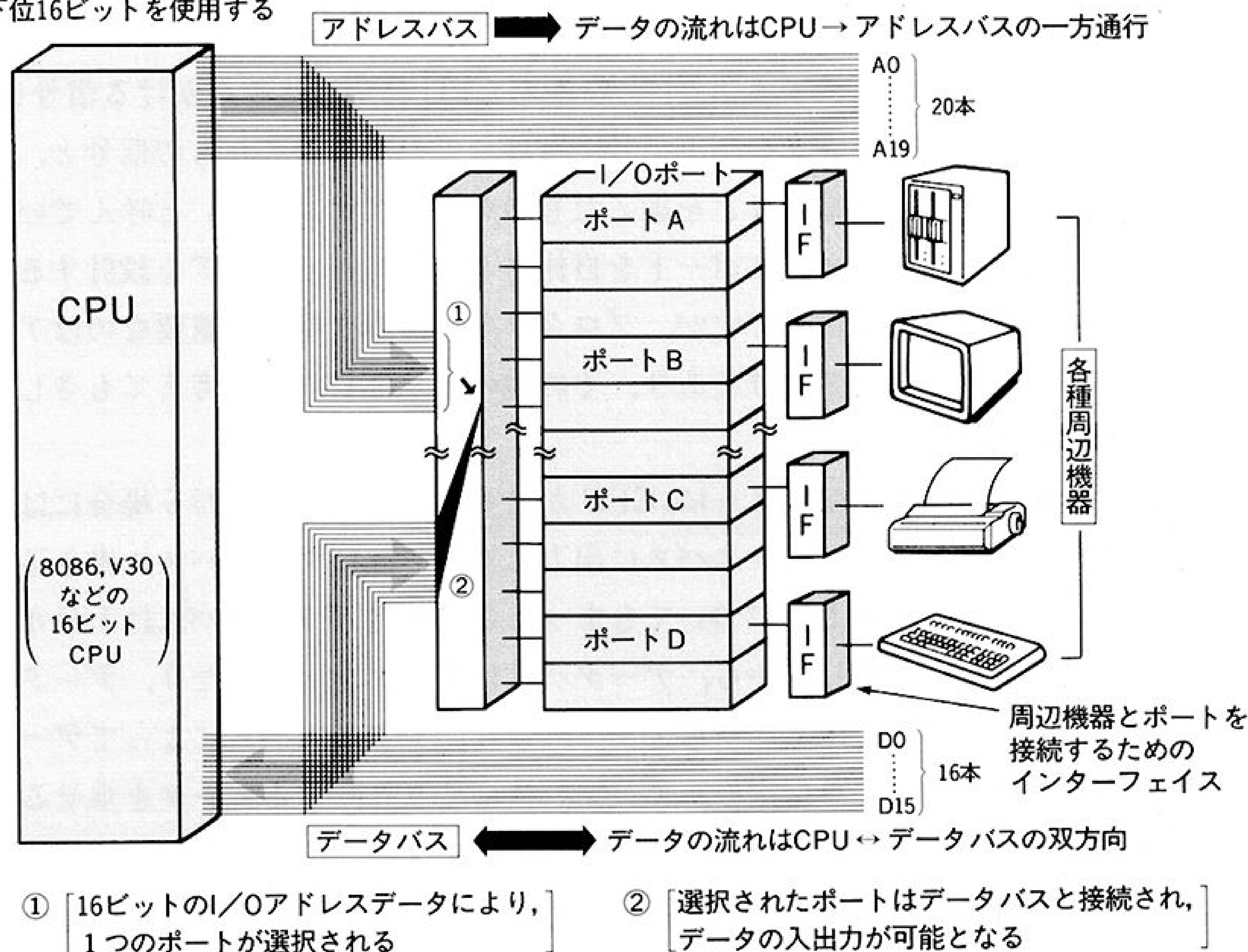


図4-19　I/Oポートからのデータの入出力

* 6809，68000など68系と呼ばれるCPU

4.6 バ　ス

CPU がメモリや I/O とデータをやり取りするための信号線の束はバス(Bus)と呼ばれます．データが乗る「乗物」のバスといった意味でしょう．データはバスに乗って CPU とメモリや I/O との間を往復するのです．主なバスには「アドレスバス」と「データバス」があります．CPU とメモリや I/O はこの 2 つのバスによってつながれています．

これ以外にも CPU とメモリや I/O は，メモリと I/O を区別する信号，読み出しと書き込みを区別する信号，I/O から CPU に割り込みをかける信号(6 章で詳述）などさまざまな制御信号で結ばれています．これらの信号と，さらに電源やクロック信号などを加えたものを「システムバス」と呼んでいます．拡張インターフェイスボードを自作するなどハードウェアを設計する場合にはどれも重要な信号ですが，プログラミングの立場から重要なのはアドレスバスとデータバスだけであり，その他の信号は省略して考えてもさしつかえありません．

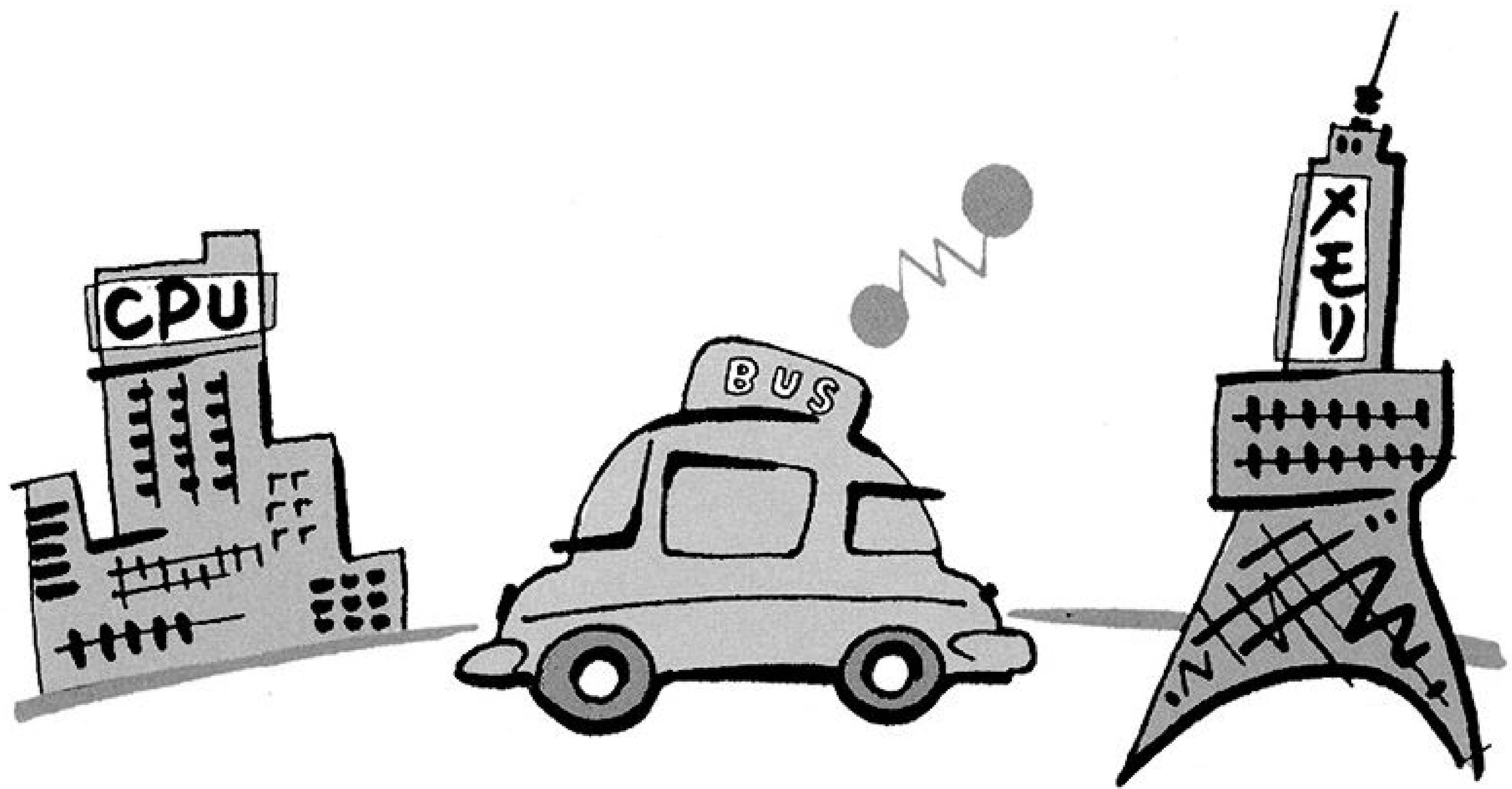
これまで何度か述べたように，CPU がメモリに書き込みを行う場合には，まずそのアドレスをアドレスバスに出力します．次にデータバスに書き込むデータを出力します．I/O についてもまったく同様で，アドレスバスに I/O ポートのアドレスを出力してから，データバスにデータを出力したり，データバスからデータを読み込んだりします．つまり，アドレスバスによってデータをやりとりする相手を指定し，データバスにやりとりするデータを乗せるのです．

バスの概念は今では当り前のことであり，マイクロプロセッサ（CPU）では必ずバスが使われています．ところが，マイクロプロセッサなどが登場する以前のコンピュータのなかには，アクセスする相手ごとに専用の信号線が用意されているものがありました．特定の I/O に対するデータは特定の信号線を通って伝わり，それ以外の I/O へのデータはその信号線を通ることはな

かったのです．いうなればバスでなくマイカーであったわけです．

バスの概念によってメモリやI/Oはアドレスやデータの乗る信号線を共有できるようになり，いろいろなI/Oへのデータが同じ信号線に乗ることによって信号線の数を大幅に減らすことができたのです．

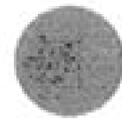
もちろん，バスといっても同時にいくつものデータが乗っているわけではなく，ある時点ではあるメモリから読み出されたデータが乗っており，次の時点ではあるI/Oへのデータが乗っている，という具合いに次から次へとお客さん（データ）が乗り降りしています．そういう意味ではバスというよりタクシーですね．

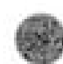


バスは信号線の数を減らすだけでなく，システムの拡張性を高めることにも役立っています．バスは特定のメモリやI/Oに依存しないため，システムに後からメモリやI/Oを追加することが簡単にできるのです．後から付けたメモリやI/Oでもバスにつなげてしまえば，もとからあるメモリやI/Oとまったく同様にアクセスすることができます．多くのパーソナルコンピュータには拡張スロットが付いており，ここにはCPUにつながるバスが引き出されています．バスを介してCPUとつなげることによって，拡張スロットのメモリやI/OにCPUがアクセスすることができるのです．

5

8086CPUの基礎





いよいよマシン語を学習する準備は整いました．前章で解説した，コンピュータ・システムの構成や仕組みといったハードウェアの知識，ビットやバイトの概念と 16 進数でデータを表現する方法などをベースに，CPU がマシン語を実行する仕組みの解説へと進んでいきます．

CPU はコンピュータ・システムの核であり，マシン語プログラムを実行することによってシステム全体を制御します．CPU の仕組みを理解することにより，マシン語が CPU に何を命令し，何を実行させるものなのかを理解することへつなげることができます．

この章では 8086CPU の仕組みを解説することを通して，マシン語とはどのようなものなのか，またマシン語によって CPU がどのような働きをするのかを具体的に解説します．

5.1 8086CPUの特徴

8086CPUについての具体的な解説に入る前に,8086CPUの特徴をまとめておくことにしましょう.

16ビットCPU

8086CPUを特徴づけることとして,まずレジスタおよびデータバスが16ビットであることが挙げられます.これは基本的なデータの処理単位が16ビットであることを意味します(ただし,8ビット単位の処理を行うことも可能です).

一度に扱うデータが16ビットであるため,8ビットまでのデータしかアクセスできない8ビットCPUに比べると,かなり高速に処理を行うことができます.また,アドレスバスを20本持っているので,8ビットCPUのメモリ空間(64Kバイト)に比較すると16倍ものメモリ空間(1Mバイト)を扱うことができます.

I/Oによる入出力

8086CPUは,メモリとは別に独立したI/O(入出力)ポートを持っています.この方式では周辺機器をI/Oポートに割り当てるだけでよいので,メモリ空間の配置に制約を加えることなく簡単に周辺機器を接続できるという特徴があります.

これに対して,68系と呼ばれる系統のCPUなどでは独立したI/Oポートを持たず,メモリ領域の特定のアドレスを周辺装置に接続して入出力を行います(メモリマップドI/O方式).この方式の利点は,メモリと周辺機器を同様の命令で扱えるということです.

1Mバイトのメモリ空間とセグメント方式

8086CPUは，20ビットからなるアドレスバスを持ち，1M（メガ）バイトまでのメモリを扱うことができます．そしてメモリ空間の中から1つのメモリを指定する方法として，「セグメント方式」を採用しています．このセグメントという考え方は，8086CPUの大きな特徴であり，8ビットCPUの資産を受け継ぐ上でも，柔軟なメモリ管理を行う上でも重要な概念です．

1Mバイトのメモリ空間をセグメントという単位に細かく区切って管理する方法は，プログラムによっては無駄を省き，高速化を実現することに役立っています．80286, 80386といった上位CPUに至るまでセグメント方式は受け継がれており，「仮想記憶」(CPUが実際にアクセスできるメモリよりも多くのメモリ空間を扱うためのテクニック）を行う上でも都合のよい方式です．

しかし，8086CPUや80286CPUでは1つのセグメントの大きさが最大64Kバイトと限定されているので，連続した大きなデータを扱うには非常に不便であることも確かです．



CPUファミリ

現在マイクロコンピュータはあらゆる分野で使われていますが，その先鞭を付けたのは8080という8ビットのCPUです．8080CPUはCP/MというOSを得て世界中に広く普及しましたが，各種のアプリケーションプログラムの規模が大きくなるにつれて処理速度やメモリ容量などの点で限界が生じてきました．この問題を解消するために8080CPUを16ビットに拡張するかたちで開発されたのが8086CPUです．このため8086CPUは，良くも悪くも多くの点において8080CPUにおける概念を受け継いでいます．

8ビットCPUからみると非常に優れた能力を持った8086CPUですが，時代の流れとともにCPUに要求される能力はますます高いものになってきており，さらに高機能な16ビットCPUや32ビットCPUが開発されています．最近いくつかのパーソナルコンピュータに登載されている80286CPUは，8086CPUを機能的に拡張するかたちで，本格的なメモリ管理と仮想記憶を可能にした16ビットCPUです．80286CPUは8086CPUの機能を完全に含んだ上でさらに機能が拡張されており（このような関係を「上位コンパチ」であ

るという), しかも処理速度がかなり高速化されているので, その使い方には, 単に速い 8086CPU として使用することも, 80286CPU 独自の機能を生かした OS によりマルチタスクを行うことも可能です.

8080 から始まるこのような CPU の系列を, 俗に 80（ハチマル）系と呼んでいます. また, 80 系の CPU は多くのパソコンで採用されたこともあって, 上位コンパチの CPU もいくつかの会社から発売されています. 8 ビットでは Z80, 16 ビットでは V30 などがその代表です.

80 系は互換性を第一に考え, これまでに開発されたハードウェアやソフトウェアおよびそのテクニックを最大限に生かすように作られています. 将来のコンピュータ界の動向を見極めることは困難ですが, 80286CPU, さらに 32 ビットへと拡張された 80386CPU へと進んでいくものならば, 8086CPU で開発したプログラムやデータ, そのテクニックは, たとえ 8086CPU 自身が使われなくなったとしても決して無駄になることはないでしょう.

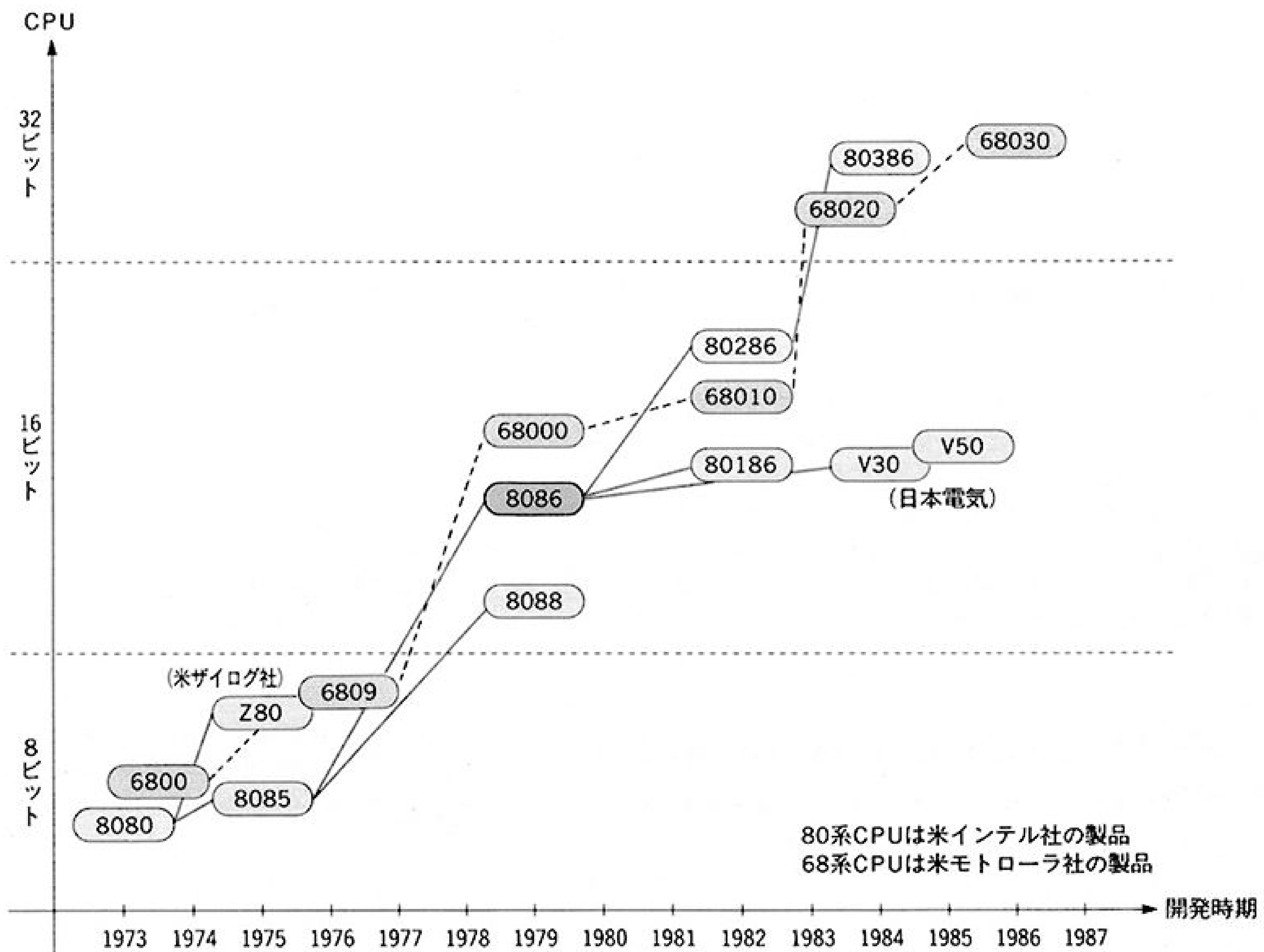


図 5-1　CPU 系統図

5.2 ニーモニックとアセンブラ

●

すでに何度も解説してきたように，CPU が直接実行できるのはマシン語だけです．CPU はメモリからマシン語命令を読み込んで解釈し実行します．この仕組みは「4.3　CPU」で解説した通りです．それでは具体的にマシン語プログラムとはどのようなものなのかを見てみましょう．

●

マシン語とアセンブリ言語の関係

マシン語命令の1つ1つにはそれぞれ異なるビットパターンが割り当てられています．私たちがビットパターンを16進数で表現することは「4.2 2進数と16進数」で解説しましたが，マシン語命令もやはり16進数で表現することができます．以下の図5-2は，メモリに格納された短いマシン語のプログラムをDEBUGでダンプしたものです．DEBUGでメモリの内容をダンプするにはD (Dump) コマンドを使います．

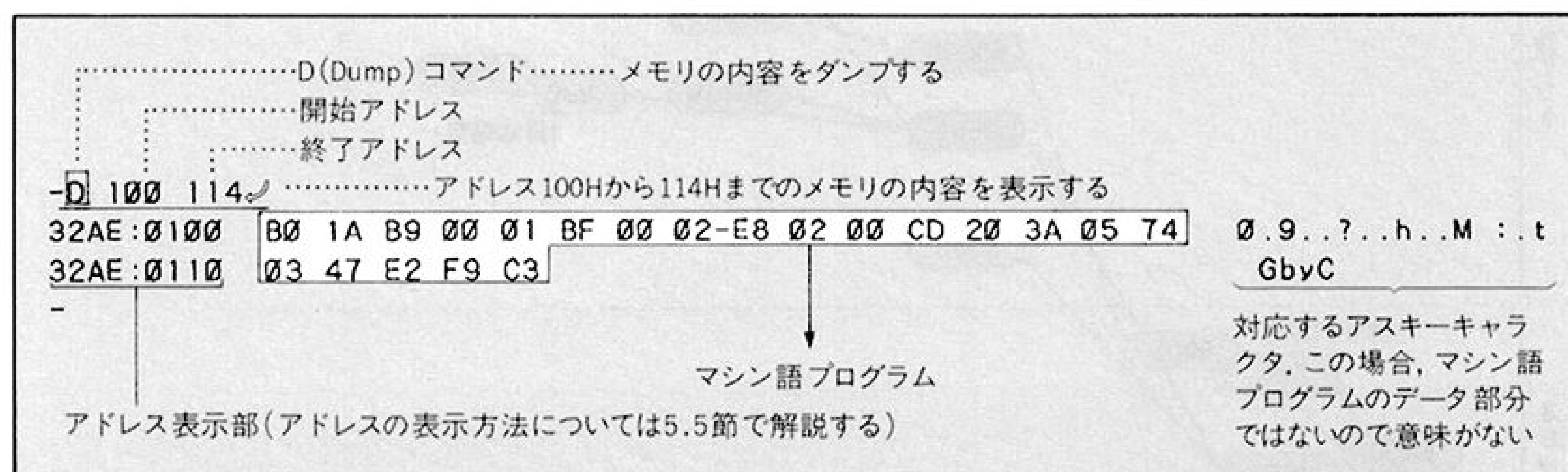


図 5-2　マシン語プログラムのダンプ

実は，これは「アドレス 200_H から 300_H の間に^Z（CTRL＋Z）のコードがあるかどうかを調べる」マシン語のプログラムなのですが（くわしくは5.6節

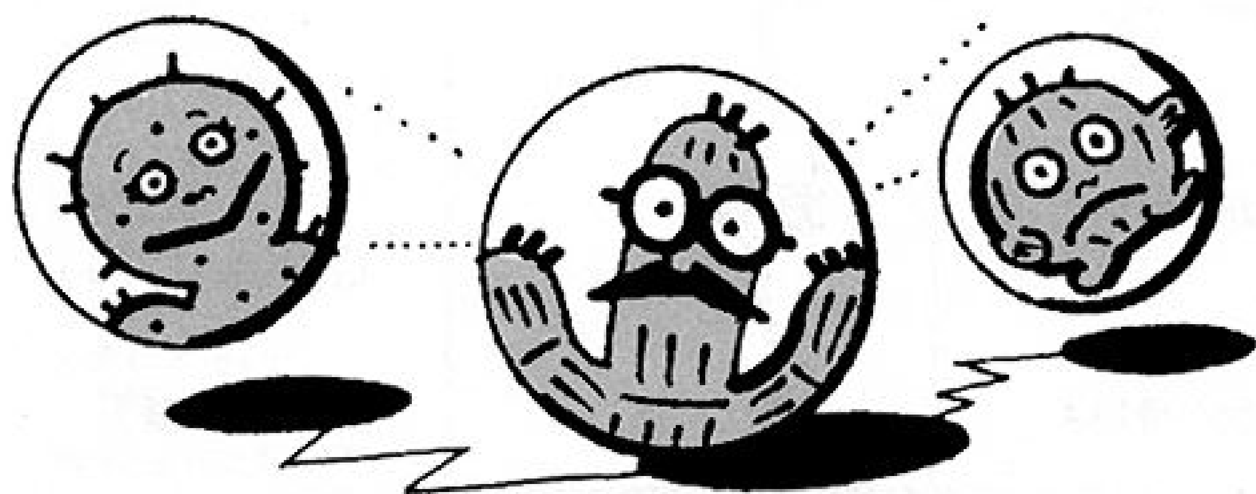
で解説する)，この16進数の羅列を眺めてもどういう内容であるかはさっぱりわかりません．8086のマシン語によほど精通している人でなければ，16進数のダンプを見ただけでその内容を理解することはできません．理解できる人でも，長いプログラムになると，その全体を把握することは不可能と言えるでしょう．

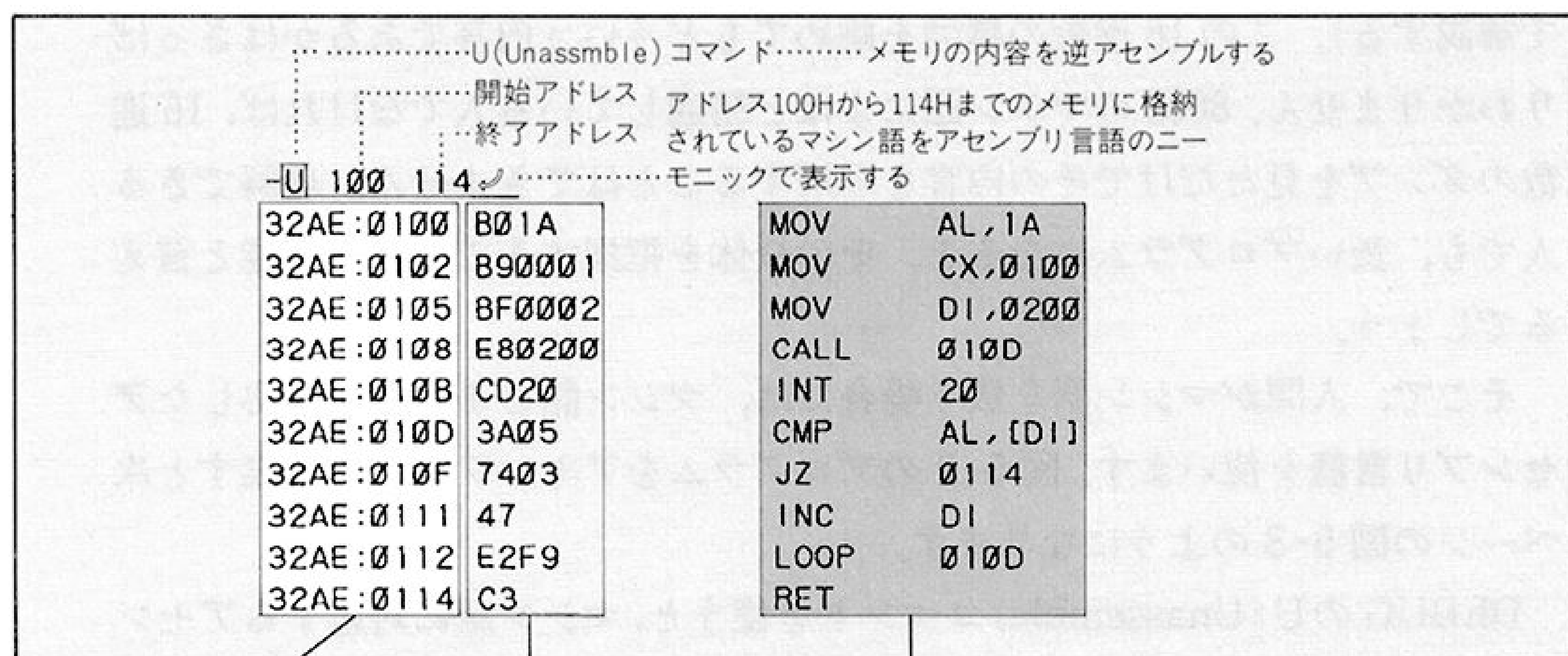
そこで，人間がマシン語を扱う場合には，マシン語と1対1に対応した**アセンブリ言語**を使います．**図5-2**のプログラムをアセンブリ言語で表すと次ページの**図5-3**のようになります．

DEBUGのU (Unassemble) コマンドを使うと，マシン語に対応するアセンブリ言語を表示することができます．**図5-3**に示すように，アセンブリ言語を使うと，16進のマシン語プログラムが意味のある言葉に近づいたことがわかると思います．

この図のようにマシン語には1つ1つ記号が割り当てられており，その記号のことを**ニーモニック**と呼びます．この記号は英語を略したもので，そのニーモニックに対応するマシン語によってCPUが行う動作を表しています．たとえば，**図5-3**の先頭にある「B01A」というマシン語は，ニーモニックの「MOV AL,1A」に対応し，〝ALレジスタに値$1A_H$を転送せよ〟というCPUに対する命令を表しています．アセンブリ言語はこのようなニーモニックの組合せを体系的にまとめたものと考えてよいでしょう．

マシン語はCPUが理解できる機械的な言語であるのに対し，アセンブリ言語は私たちが理解できる人間の言葉に近い言語です．私たちは通常，このアセンブリ言語を使ってマシン語プログラムを記述します．一般に，「マシン語でプログラムを書く」とか「マシン語がわかる」とかいう場合には，このアセンブリ言語でプログラムを書くことや，アセンブリ言語で書かれたプログラムの内容がわかるということを意味します．





アドレス	マシン語 命令	マシン語 パラメータ	ニーモニック オペコード(主命令)	ニーモニック オペランド(主命令が対象とする相手, 条件, パラメータ)
32AE:0100	B0	1A (メモリに格納されている順序)	MOVe(ムーブ) MOV 転送せよ	AL,1A ALレジスタに値1AHを
32AE:0102	B9	0001	MOVe(ムーブ) MOV 転送せよ	CX,0100 CXレジスタに値0100Hを
32AE:0105	BF	0002	MOVe(ムーブ) MOV 転送せよ	DI,0200 DIレジスタに値0200Hを
32AE:0108	E8	0200	CALL(コール) CALL サブルーチンを呼べ	010D アドレス010DHの
32AE:010B	CD	20	INTerrupt(インタラプト) INT 内部割り込みをかけよ	20 20番の
32AE:010D	3A	05	CoMPare(コンペア) CMP 比較せよ	AL,[DI] ALレジスタとDIレジスタの指すアドレスのメモリの内容を
32AE:010F	74	03	Jump if Zero(ジャンプイフゼロ) JZ ゼロフラグが1ならばジャンプせよ	0114 アドレス0114Hへ
32AE:0111	47		INCrement(インクリメント) INC レジスタDIを1増加せよ	DI
32AE:0112	E2	F9	LOOP(ループ) LOOP CXの値を1減らし，0でなければジャンプせよ	010D アドレス010DHへ
32AE:0114	C3		RETurn(リターン) RET サブルーチンから復帰せよ	

図 5-3　図 5-2 のマシン語プログラムをアセンブリ言語で表すと…

アセンブルと逆アセンブル

16進のマシン語とアセンブリ言語のニーモニックは1対1に対応していますから，「マシン語をアセンブリ言語へ」，逆に「アセンブリ言語で書かれたものをマシン語へ」と機械的に変換することができます．アセンブリ言語のニーモニックからマシン語に変換する作業を**アセンブル**と呼びます．DEBUGではA (Assemble) コマンドを使うことによってアセンブルを行うことができます．このコマンドは，アセンブリ言語のニーモニックを1行ずつ入力するごとに，それに対応するマシン語に変換し，メモリに格納していきます．図5-4にその例を示してみましょう．

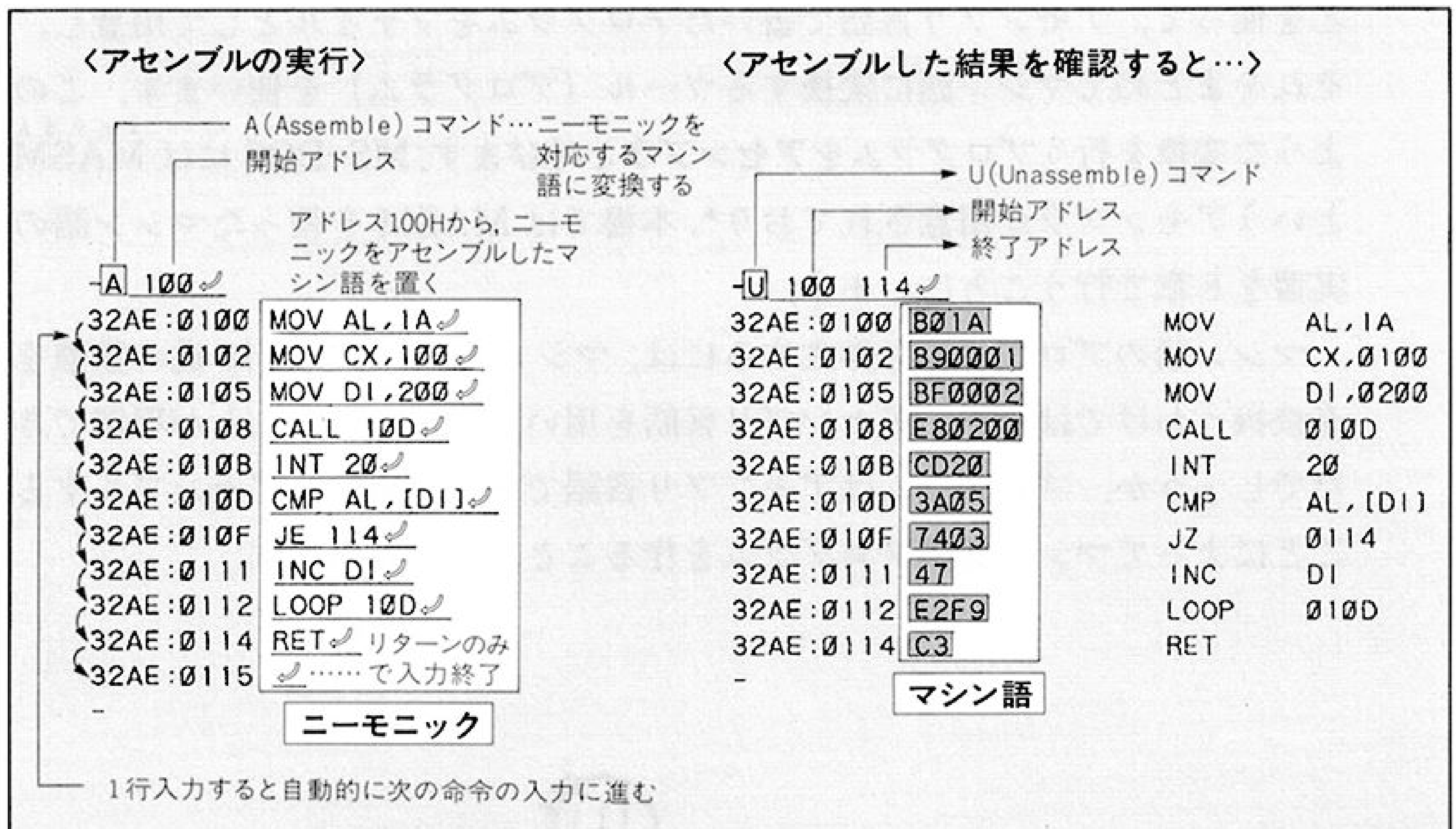


図5-4　Aコマンドによるアセンブル

先の図5-3ではDEBUGのUコマンドを使って，マシン語をアセンブリ言語のニーモニックに変換しました．これはアセンブルの逆の操作です．この操作を**逆アセンブル**あるいは**ディスアセンブル**と呼びます．

マシン語プログラムを作成する道具(ツール)

かつて8ビットマシンしかなかった頃，アマチュアの間ではハンドアセンブルといってアセンブリ言語とマシン語の対応表を一生懸命めくりながら，マシン語のコードを調べてマシン語プログラムを書いていたものですが，MS-DOSではDEBUGの助けを借りることによって簡単にマシン語のプログラムを作成することができます．マシン語の勉強に特別な道具はいりません．MS-DOSのシステムと，それに含まれているDEBUGコマンドさえあればよいのです．

DEBUGのAコマンドによるアセンブルは一度入力したら修正できないなど，その場限りのものですが，本格的なプログラムの作成には，エディタなどを使って，アセンブリ言語で書いたプログラムをファイルとして用意し，それをまとめてマシン語に変換するツール（プログラム）を使います．このような変換を行うプログラムを**アセンブラ**と呼びます．MS-DOSにはMASM(エムアセム)というアセンブラが用意されており*，本書ではMASMを使ったマシン語の実習を8章で行うことにします．

マシン語のプログラムを作成するには，マシン語，すなわち16進の数値を直接扱うわけではなく，アセンブリ言語を用いるのだということが理解できたでしょうか．プログラムはアセンブリ言語で書き，それをアセンブルすることによってマシン語のプログラムを作ることができるのです．



*メーカーによっては，MASMはMS-DOSのシステムディスクには含まれず，別売されている場合もある．

5.3 レジスタとその機能

4.1章で示したように，メモリはバスを介してCPUにつながれています（55ページの図4-1参照）．このメモリとは別に，CPUの内部にもいくつかのメモリが存在し，「レジスタ」(Register)と呼ばれています．レジスタには8ビットまたは16ビットのデータを記憶することができ，1バイトまたは1ワードのメモリとまったく同等です．ただしレジスタはバイトやワードといった単位では数えません．レジスタは1本2本と数えるのが普通です．

また，メモリには1バイトごとにアドレスが割り当てられており，アドレスで1つ1つを区別しますが，レジスタにはすべて名前（レジスタ名）が付けられており，アドレスではなく名前で区別します．

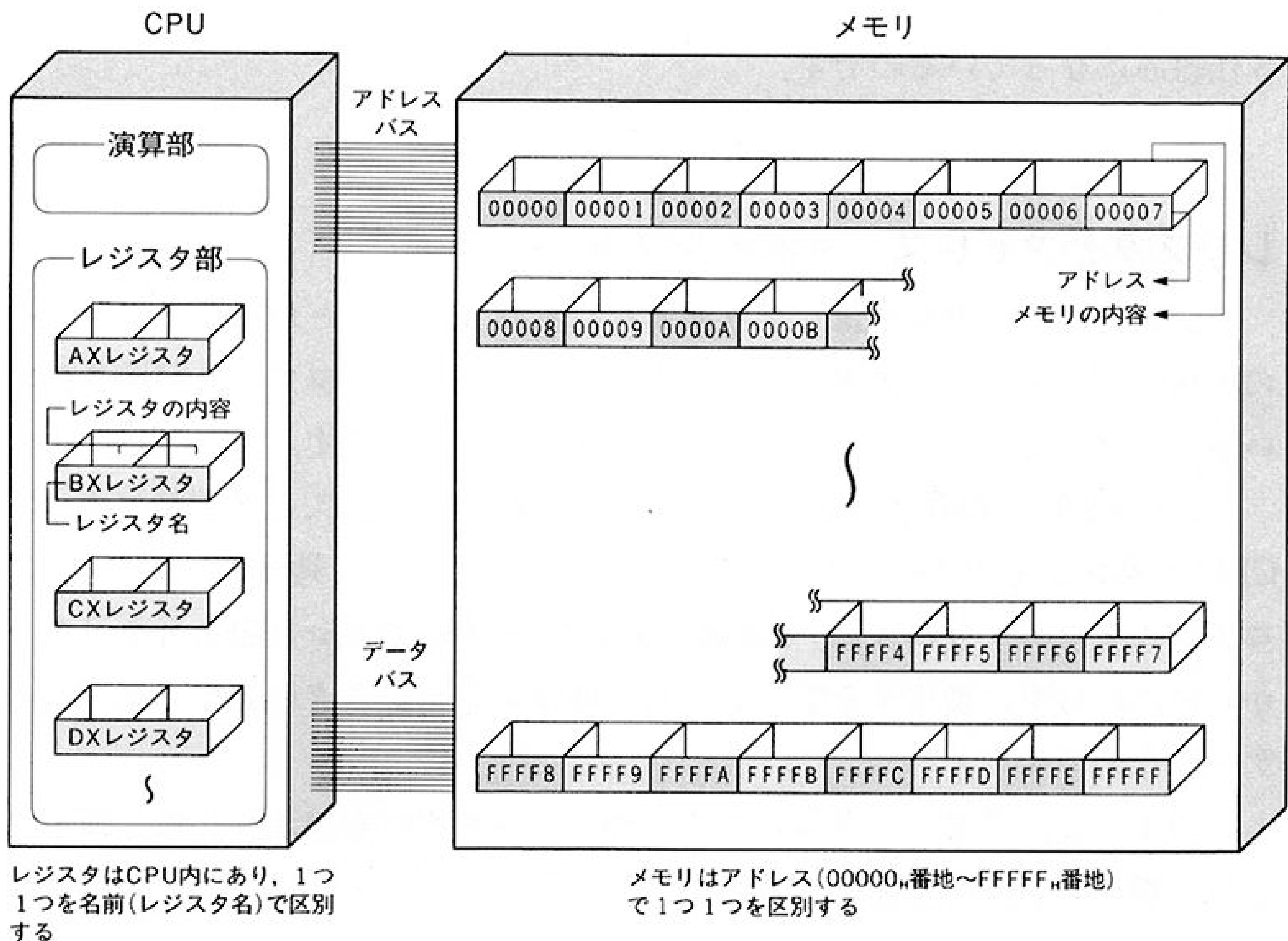


図5-5 レジスタとメモリの違い

レジスタとマシン語プログラム

マシン語の命令の多くはレジスタを操作する命令であり，マシン語プログラムの主要な動作はレジスタを操作することに費やされます．レジスタの基本的な操作とは，具体的には次の3つです．

1．**メモリやI/Oからレジスタにデータを転送する．**
2．**レジスタに記憶されているデータに対して演算を行う．**
3．**レジスタからメモリやI/Oにデータを転送する．**

マシン語のプログラムは，最終的にはたったこれだけのことの組合せでしかありません．どんなに複雑な処理でもこの3つの操作をいろいろと組み合わせることによって実現しているのです．「レジスタを使う」ことがマシン語のプログラムであるということを覚えておいてください．

レジスタはCPUの内部にあるため，メモリ上のデータを操作するときに必要なアドレスの指定やバスへのデータの出力などの一連の手順を踏む必要がなく，データの転送や演算を高速に行うことができます．データを高速に処理する必要から，いったんCPU内部のレジスタに読み込んで処理を行うという仕組みになっているのです．

●

レジスタとアドレス　―ポインター

レジスタはデータを記憶したり演算を行ったりすることができますが，その応用としてもう1つ大事な機能があります．それはレジスタに記憶されているデータによってメモリのアドレスを指定することです．

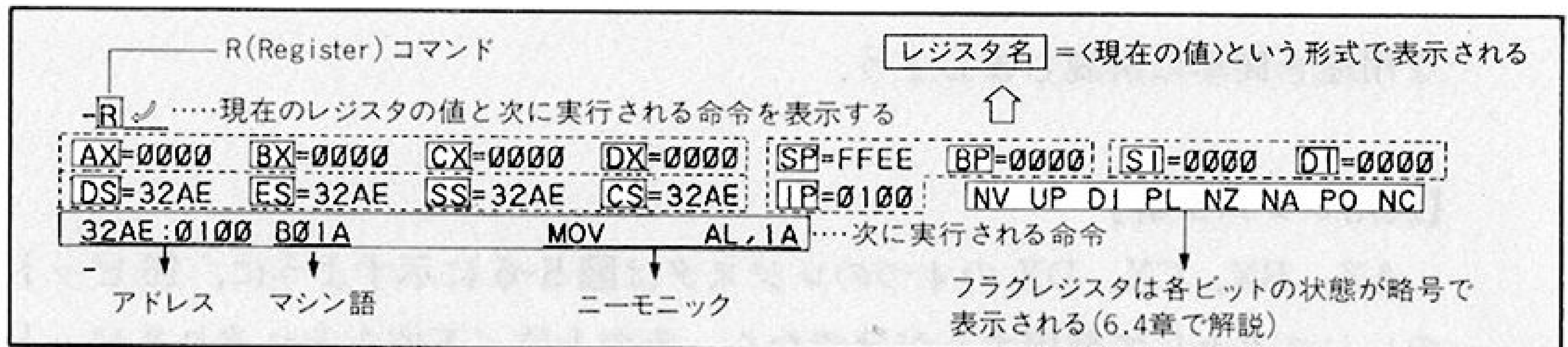
これはCPUの動作のメカニズムにとって重要なことで，この機能によってCPUの動作が実現されるといっても過言ではありません．具体的には「5.6　プログラム実行のメカニズム」で解説しますが，CPUがマシン語命令をメモリから読み込む際に指定するアドレスも，実はあるレジスタに記憶されているデータなのです．

このように，アドレスを指定するためにレジスタに記憶された値を使うことを，**ポインタ**として使うと言います．

8086CPUのレジスタ構成

8086CPUに用意されているレジスタを次の図5-6に示します。8086CPUには16ビットのレジスタが14本用意されており，それぞれ図のように名前が付けられています。

なお，この図でわかるようにDEBUGのR(Register)コマンドを実行することにより，すべてのレジスタの現在の値を表示することができます。



CPU

●汎用レジスタ群●

15 AX 0 アキュムレータ 7 AH 0 7 AL 0

BX ベースレジスタ BH BL

CX カウントレジスタ CH CL

DX データレジスタ DH DL

●インデックスレジスタ群●

15 SI 0 ソースインデックス

DI ディスティネーションインデックス

●特殊レジスタ群●

15 BP 0 ベースポインタ

SP スタックポインタ

IP インストラクションポインタ

●セグメントレジスタ群●

15 CS 0 コードセグメント

DS データセグメント

ES エクストラセグメント

SS スタックセグメント

●フラグレジスタ●

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

OF DF IF TF SF ZF AF PF CF

図5-6　8086CPUのレジスタ

各レジスタには名前のほかに2文字の英文字からなる略号が付けられています．たとえば「アキュムレータ」という名前のレジスタがありますが，その略号が「AX」であり，通常は略号の方を使って「AXレジスタ」と呼びます．他のレジスタについても同様です．アセンブリ言語でレジスタを指定する場合も「AX」のように略号の方を用います．

各レジスタにはそれぞれ決まった用途があり，その種類によって図5-6のようなグループに分けることができます．各グループについてレジスタの主な用途を簡単に解説しましょう．

【汎用レジスタ群】

AX，BX，CX，DXの4つのレジスタは図5-6に示すように，16ビットのレジスタとして使用するだけでなく，その上位／下位をそれぞれ8ビットのレジスタとして単独で使用することもできます．たとえばAXレジスタの上位8ビットには「AH」，下位8ビットには「AL」という略号が付けられています．Hは上位を表す「High」，Lは下位を表す「Low」を意味しており，同様にBXは「BHとBL」，CXは「CHとCL」，DXは「DHとDL」という2本の8ビットレジスタとして使うことができます．

これらのレジスタはデータを記憶しておいたり，種々の演算に使用することのできる汎用レジスタですが，各レジスタとも特有の機能を持っており，それらのレジスタと組み合わせて使う命令が決っています．それぞれの具体的な使い方に関しては6章で解説，実習しますので，ここでは各レジスタに固有の機能について簡単に表にしてまとめておきます（表5-1）．

レジスタ	名　称	固有の機能
AX	アキュムレータ	・各種の演算（他のレジスタより高速） ・乗除算
BX	ベースレジスタ	・特定のメモリを指し示すポインタ
CX	カウントレジスタ	・転送や繰り返しの回数を数えるカウンタ
DX	データレジスタ	・データの一時記憶用 ・AXと組み合せて，32ビットの乗除算

この4つのレジスタは汎用レジスタであるので，もちろん加減算，比較，論理演算なども行うことができる．
各レジスタの使用法は6章を参照のこと．

表5-1　汎用レジスタの機能

【インデックスレジスタ群，特殊レジスタ群】

これらのレジスタは，IPレジスタを除いて，汎用レジスタと同じような機能を持っています．しかし汎用レジスタではデータの記憶や演算そのものが目的であるのに対し，これらのレジスタはいずれもメモリのアドレスを指定するためのポインタとして使われます．たとえば，SIレジスタの内容をアドレスとするメモリの内容をAXレジスタに転送する，という使い方ができるのです．具体的には6章で解説，実習を行います．

また，SPおよびIPレジスタは，プログラムで直接その値を利用してアドレスを指定するということはできませんが，プログラムの実行などを制御するためのレジスタとして間接的に利用することになります．

【セグメントレジスタ群】

これらのレジスタもメモリのアドレスを指定するために使われます．ただし，すでに紹介したポインタの機能を持つレジスタとは異なる意味を持っており，セグメント方式と呼ばれるメモリ管理方式と深い関係があります．くわしくは「5.5 セグメントの考え方」で解説します．

【フラグレジスタ】

このレジスタは他のレジスタとは異なり，データを一時的に蓄えたり，データを操作する目的に使うことはできません．フラグレジスタは，プログラムで操作するのではなく，プログラムの実行によって変化するCPUの状態を表すために専用に用意されているレジスタなのです．

CPUの状態として重要なのは演算の結果の状態であり，たとえば「加算や減算などの各種の演算の結果が正か負か」，「演算の結果がオーバーフロー(桁あふれ）したかどうか」などがあります．そして，フラグレジスタ中のビット1つ1つがそういった状態を表すように割り当てられており，その値が「0」であるか「1」であるかによってどちらの状態であるかを表します．演算命令などを実行すると，その結果に応じて自動的に対応するビットが変化します．それを旗が上がることにたとえてフラグと呼ぶのです．フラグレジスタのあるビットが1にセットされることを「フラグが立つ」とも言います．

このフラグを利用することによって，条件判断などを行いプログラム実行の流れを変えることができます．くわしくは6.4章で実習解説します．

5.4 スタックとその働き

この節では，プログラムを実行する際のメカニズムとして非常に重要な概念であるスタックについて解説します．

スタックとは

マシン語のプログラムでは，処理するデータや処理に必要なデータを格納するためにメモリ領域を必要とします．このようなメモリ領域のことを「ワークエリア」（作業領域）と呼びます．ワークエリアは，必要なデータ量をプログラムを作成する時点で確保しておかなければなりません．

ところがプログラムを作成する上では，ほんの一時だけデータを格納するメモリ領域を必要とする場合が多く出てきます．たとえば，「あるレジスタを一時的に演算のために使いたいが，現在のレジスタの内容は保存しておいて演算が終わった後で再びもとの状態に戻したい」というような場合です．こういったことはマシン語のプログラムを組む際に頻繁に起こりますから，個々の場合についてそれぞれ値を保存するワークエリアを確保すると膨大な量になってしまいます．これらのメモリ領域はプログラムの部分部分で一時的にしか使われないわけですから，同じメモリ領域を共有すればメモリの使用効率をよくすることができます．

そこで，生まれたのが「スタック」という概念です．スタックを使えば，一時的にデータを格納するためのメモリ領域を確保でき，格納したデータを取り出すことによってその領域を解放するということが自動的にできます．スタックを実現するために使われるメモリ領域全体を「スタックエリア」，または単に「スタック」と呼びます．スタックの概念において肝心なことは，データを格納する際にそのメモリ領域のアドレスをまったく意識する必要がないということです．

スタックの仕組み

では次にスタックがどのような仕組みで実現されているかを解説しましょう．スタックは，本を積み上げることによく似ています．このためスタックにデータを格納することを「スタックに積む」と言います．スタックにいくつかのデータを積んだとすると，最初に取り出せるのは最後に積んだデータです．これは積み上げた本の中から，一番上の本，つまり最後に積んだ本を取り上げるのと同じことです（図 5-7）．

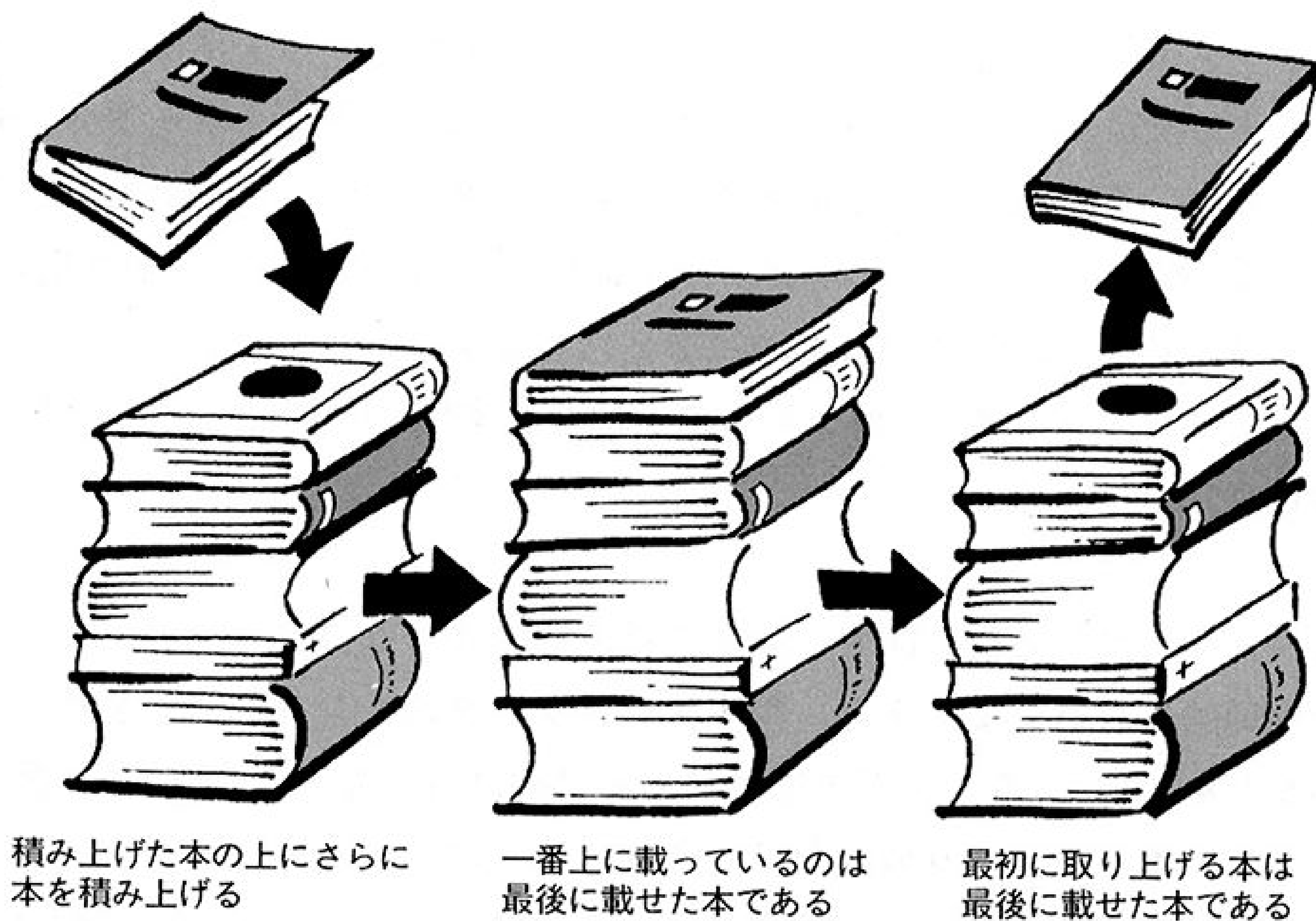


図 5-7　スタックの概念

このスタックを実現するために用意されているのが，スタックポインタ(SP)というレジスタです．ポインタと呼ばれることからもわかるように，スタックポインタはスタックエリアのメモリを指し示す役割を持っており，常にスタックトップ，つまり一番上に積んである本を指しています．

プログラムや通常のデータがアドレスの低い方から高い方へと順番に並んでいるのに対し，スタックポインタの指すアドレスはデータを積み上げるにつれて逆にアドレスの高い方から低い方へと進んでいきます．スタックにデータを積むとスタックポインタの値は減らされ，次にデータを積むアドレスはさらに低いアドレスになるのです．

スタックのデータの出し入れは１ワード，つまり２バイト単位で行われます．スタックにデータを積むときにはスタックポインタは１ワード分自動的に減らされ，そこにデータが格納されます．スタックからデータを取り出すときには，スタックポインタの指しているアドレスからデータが取り出され，スタックポインタに値２が自動的に加えられます．この様子をAXレジスタの内容をスタックに積み，再びその値を取り出す場合を例にとって図5-8に図示してみましょう．

スタックの出し入れに関する命令では，このようなスタックポインタの操作が自動的に行われますから，現実にはスタックポインタがどこを指しているかなどを，プログラミングの際に考える必要はほとんどありません．図5-7のように，本を積んだり取り上げたりするように入れたり出したりすることができ，入れた順番とは逆の順番で，つまり最後に入れたものから順に出てくる，ということだけを理解していればよいのです．

このような単純な仕組みであるにもかかわらず，スタックには実にさまざまな使い道があります．プログラムで必要な値を一時的に保存するばかりでなく，プログラム実行のメカニズムにも利用されているのです．

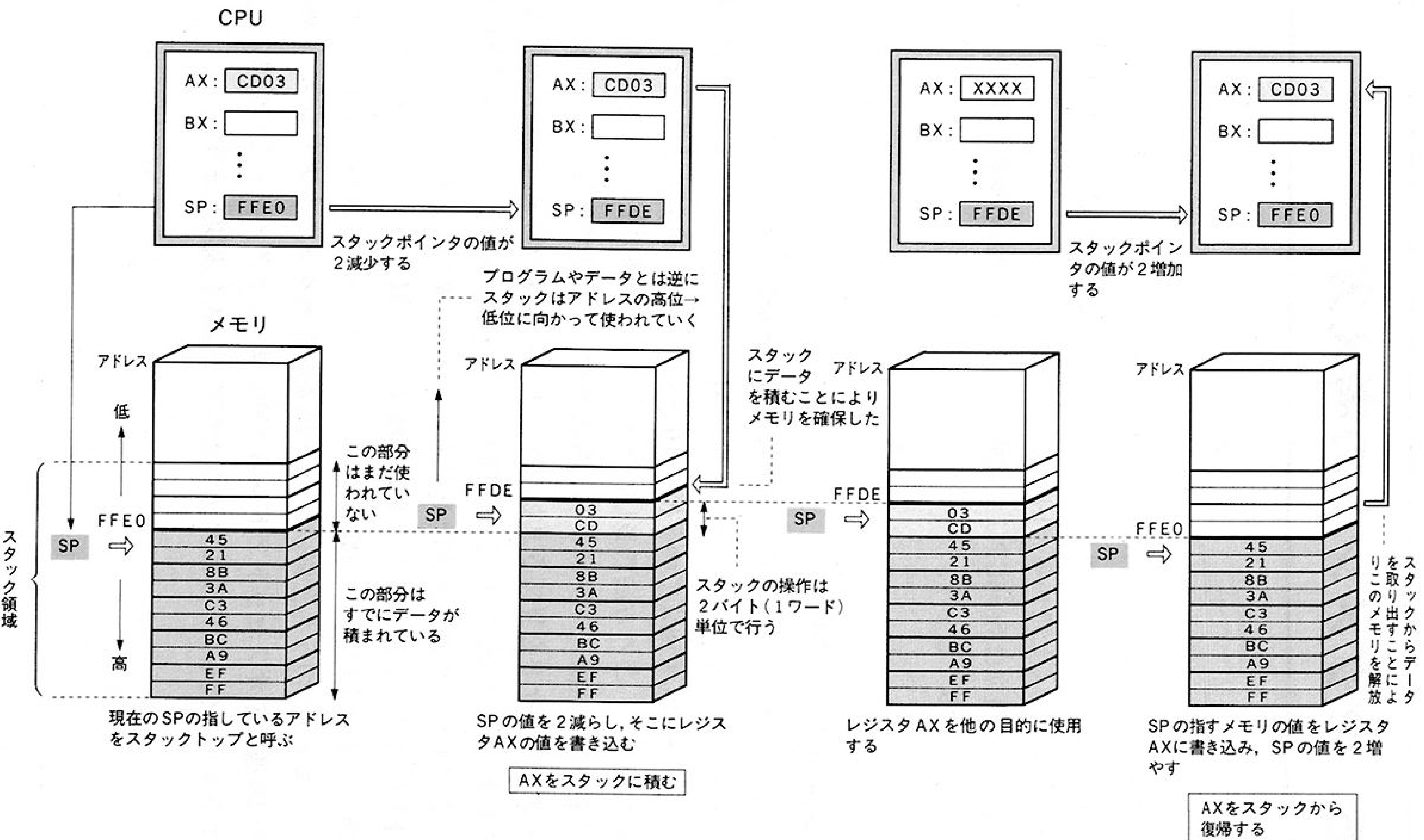


図 5-8　スタックの出し入れ

5.5 セグメントの考え方

5.1 節で解説したように，8086CPU の大きな特徴にセグメント方式によるメモリ管理があります．これは 8 ビット CPU にはなかった概念であり，8086CPU の理解をむずかしくしている一因でもあります．

セグメントの概念は 8086CPU を理解する上で欠かせない重要な概念であり，1 つの鍵と言えるでしょう．

アドレスの指定方法

8086CPU には，アドレスを指定するための 2 種類の考え方があります．1 つは「4.4 メモリ」の項で解説したように，1M バイトのすべてのメモリを 00000_H 番地から $FFFFF_H$ 番地までの 5 桁の 16 進数（つまり 20 ビットの数値）で指定する方法です．この方法で表したアドレスを「物理アドレス」と呼びます．

もう 1 つの方法は「セグメント」方式です．これまで何度となく DEBUG のダンプリストなどを見てきましたが，アドレス部の表現については何も解説してきませんでした．このことに疑問を抱いた方もいるでしょう．実は DEBUG におけるアドレスの表示方法はセグメント方式に基づくものなのです．

DEBUG のリストにおけるアドレス部は，次の図 5-9 のように 2 つの4桁の 16 進数（つまり 16 ビットの数値）を「：」（コロン）で区切って並べた形式をしています．このうち左側の 4 桁の 16 進数を**セグメントアドレス**，右側の 4 桁の 16 進数を**オフセットアドレス**と呼びます．セグメント方式ではこの 2 つのアドレスを使って，1 つのメモリのアドレスを表現します．

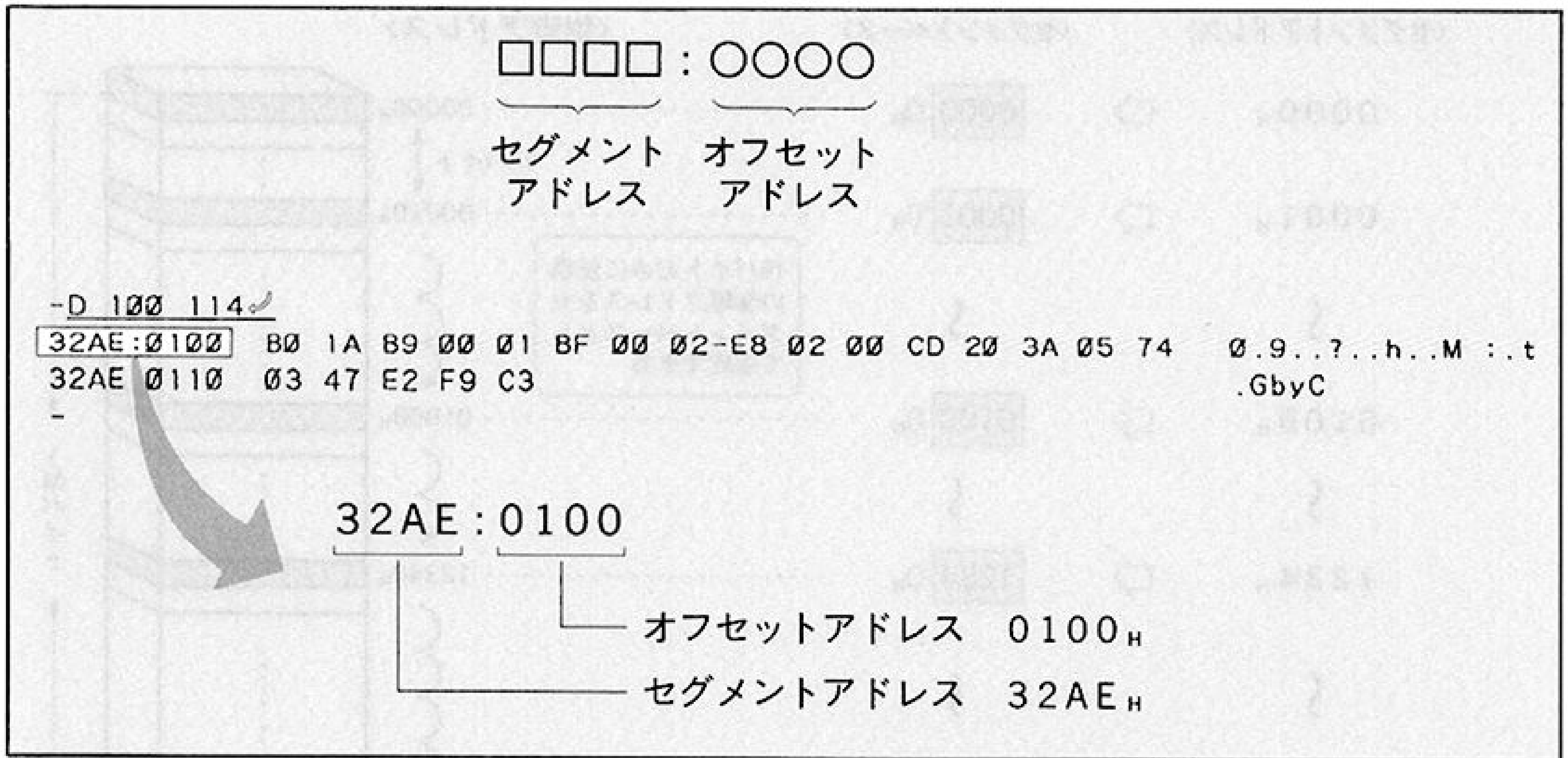


図5-9　セグメントアドレスとオフセットアドレス

マシン語のプログラミングでは物理アドレスを直接指定することはできません．物理アドレスはハードウェア的なレベルでメモリを考える場合に用いるもので，プログラムでは必ずセグメント方式のアドレス指定を行います．では，セグメント方式によるアドレスの表現方法は物理アドレスにどう対応するのでしょうか．

●

セグメントアドレスと物理アドレス　ーセグメントベースー

セグメントアドレスとオフセットアドレスという2つの16ビットの値と物理アドレスとの対応関係を解説するために，まずセグメントアドレスと物理アドレスの関係から説明しましょう．

セグメントアドレスは，「セグメントベース」という，一種の基準点となる物理アドレスを決めるための値です．次の図5-10を見てください．

このように，セグメントアドレスの最後に「0」を付け加え5桁のアドレスを作ることで，1Mバイトのアドレス空間の中から16バイトおきに任意の物理アドレスをセグメントベースとして指定できるのです．

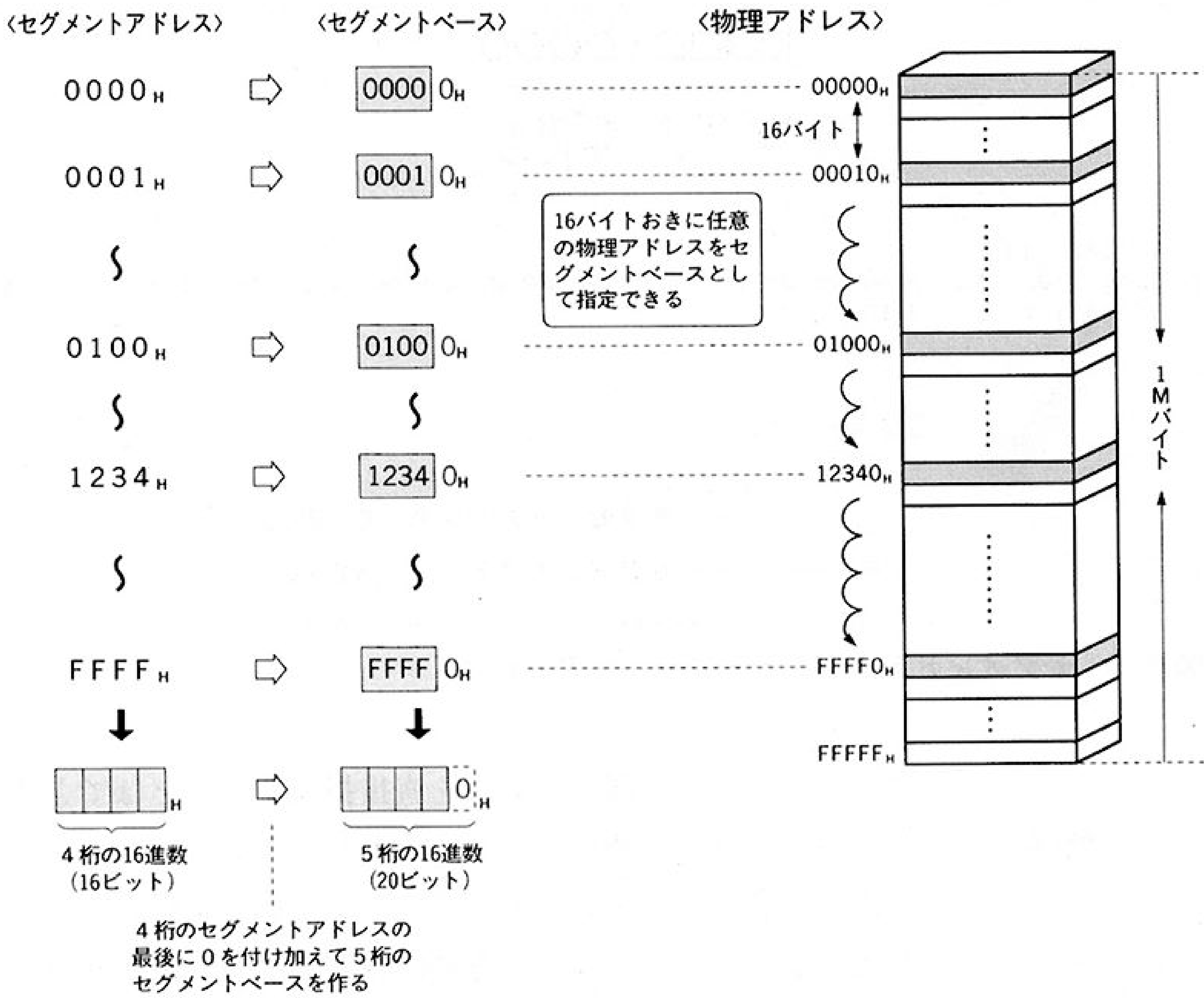


図5-10　セグメントアドレスとセグメントベース

オフセットアドレスとセグメント

セグメント方式のアドレス指定は，セグメントベースを基準点とした新たなアドレス空間を想定することによって行います．全メモリ空間からセグメントベースを起点とするメモリ領域を抜き出して，その中でのアドレスを考えます．抜き出したメモリ空間のことを「セグメント」と呼び，セグメント内部のメモリに新たに付けたアドレスのことをオフセットアドレスと呼びます．この関係を図5-11に示します．

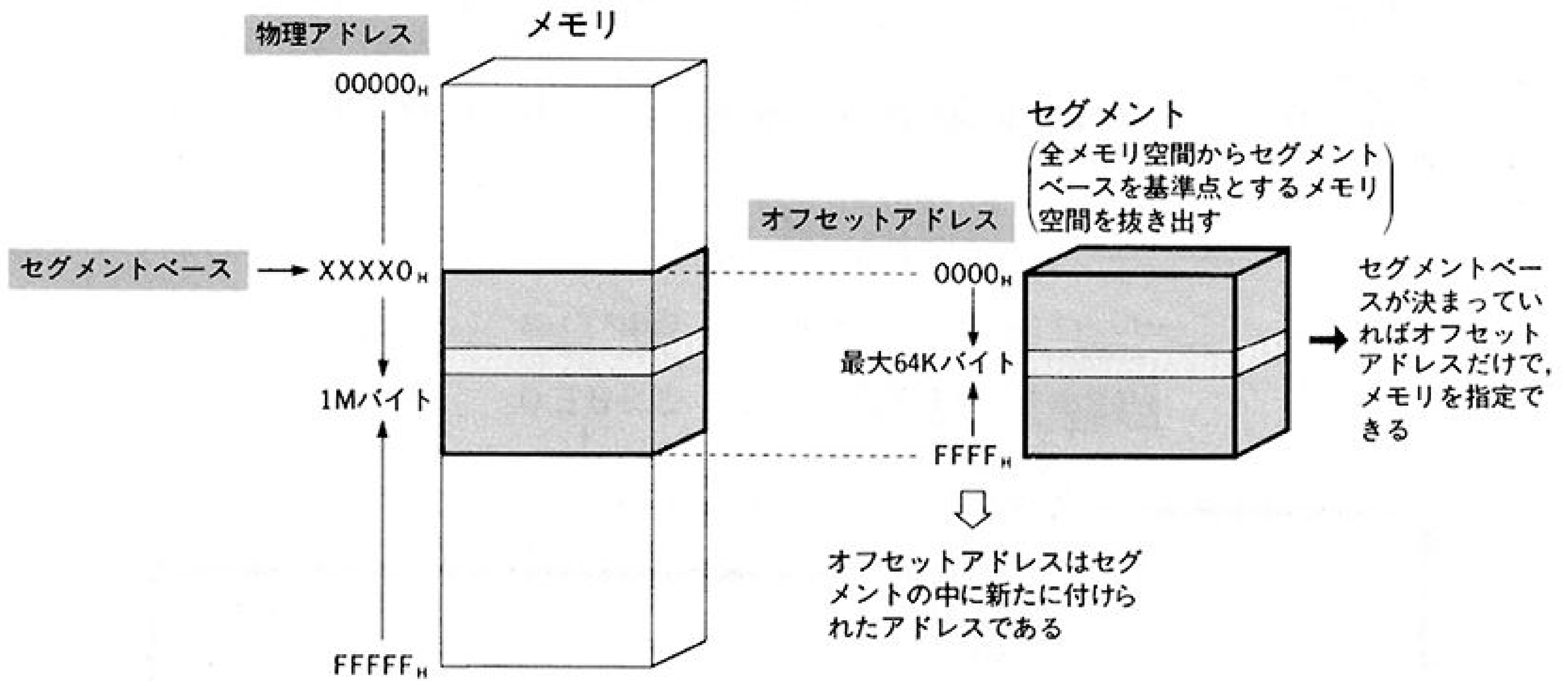


図 5-11　セグメントとオフセットアドレス

オフセットアドレスは16ビットの値なので，0_Hから$FFFF_H$までの数値となります．したがって1つのセグメントは最大64Kバイトの大きさを持つことができます．ここで「最大」というのは64Kバイト以下の大きさを持つとも考えることができるからで，その意味するところについては後の項で解説します．

セグメントベースを固定して考えるならば，オフセットアドレスだけでアドレスを指定することができます．このことを利用してセグメントアドレスは固定しておいて，そのセグメントの中での処理ではオフセットアドレスだけをアドレスとして扱おうとするのがセグメント方式の考え方です．

セグメント方式と物理アドレス

ここまでくればセグメントアドレスおよびオフセットアドレスと物理アドレスとの関係は簡単にわかるでしょう．セグメントアドレスによって決まるセグメントベースにオフセットアドレスを加えた値が物理アドレスになるというわけです．

具体的な例でこの関係を図示しておきましょう．

```
-D 100 114
32AE:0100  B0 1A B9 00 01 BF 00 02-E8 02 00 CD 20 3A 05 74   0.9..?..h..M :.t
32AE:0110  03 47 E2 F9 C3                                    .GbyC
-
```

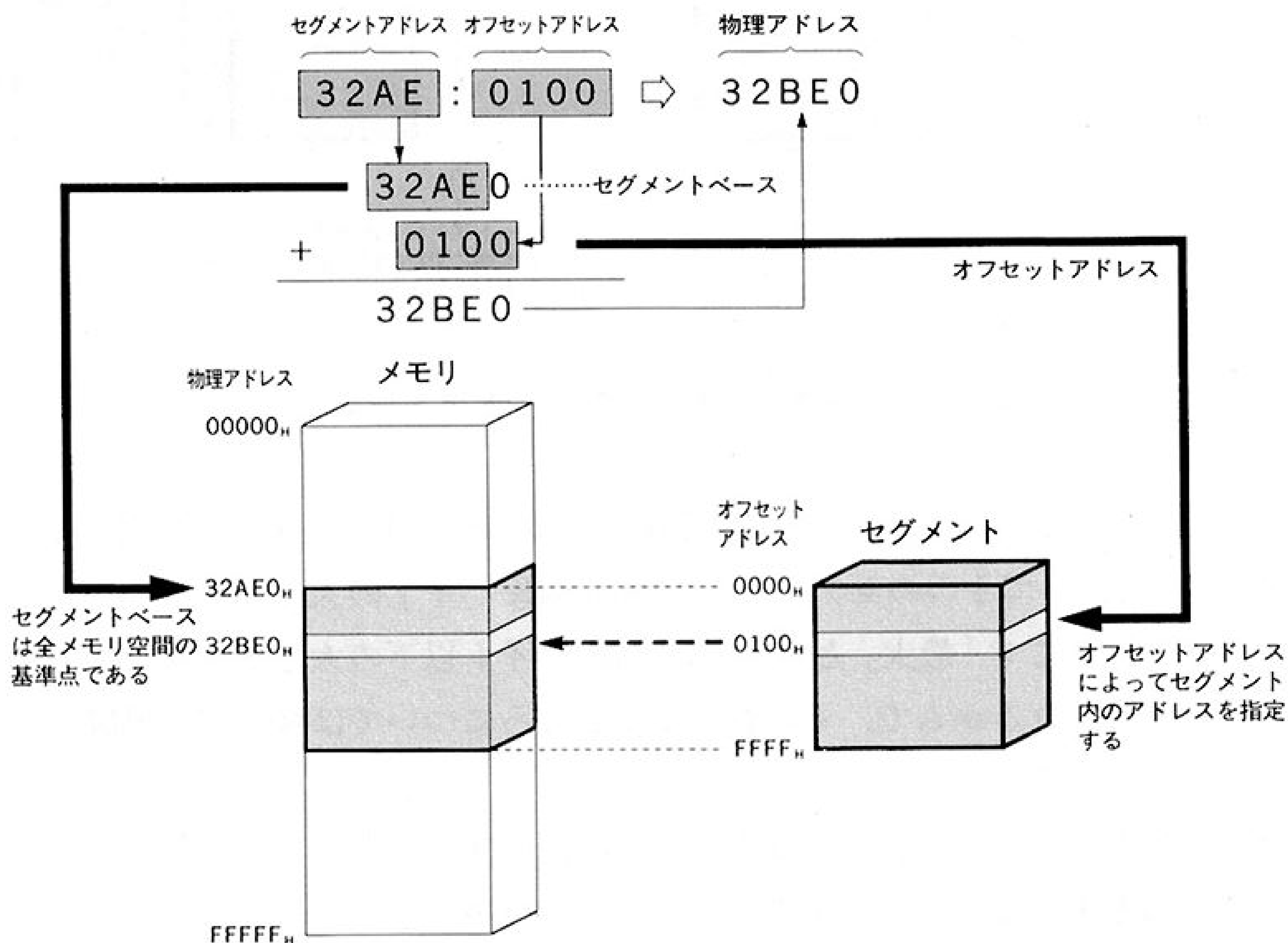


図 5-12　セグメント方式と物理アドレス

ところで，全メモリ空間からは図 5-12 のようなメモリ領域をいくつも抜き出すことができます．つまりメモリ内にいくつものセグメントを設定できるのです．マシン語プログラムでは物理アドレスを直接扱えませんから，セグメントを設定してプログラムやデータをメモリ上に配置しなければなりません．セグメントの大きさは 64K バイト以内であれば，プログラムやデータのそれぞれの大きさに合わせてそのセグメントを扱うプログラムで設定することができます．こうしてメモリ上にいろいろな大きさを持ったセグメントを配置することができます．

セグメント方式は広大なメモリ領域をこのようにしていくつもの小さな領域に分割して管理するための方法です．メモリ上にいくつものセグメントが配置されている様子を次の図5-13に示します．

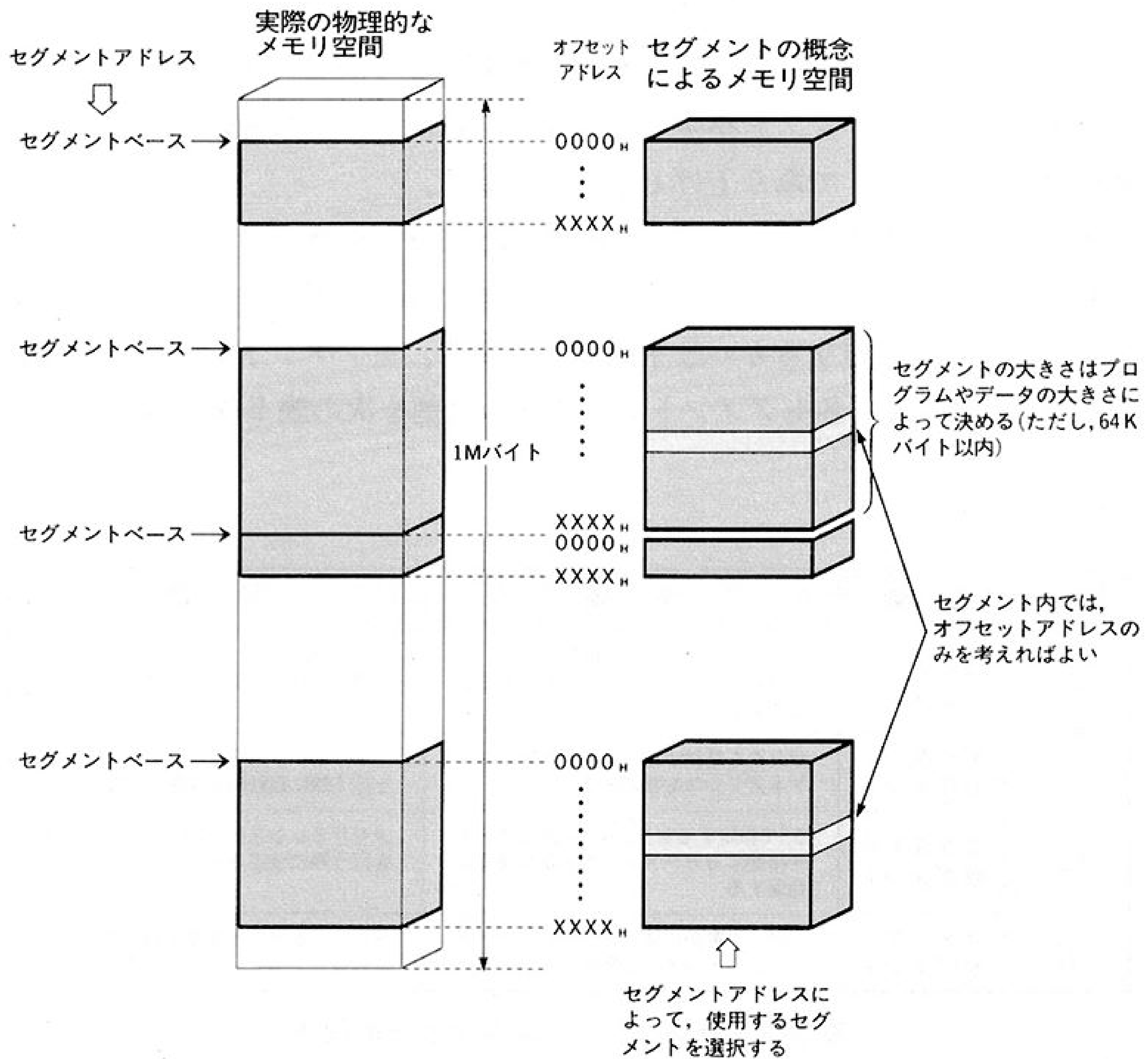


図5-13　メモリはセグメントの集まり

セグメントレジスタ

「5.3　レジスタとその機能」で解説したように，CPUのレジスタはメモリのアドレスを指定するためにも使用されます．アドレスを指定するためには16ビットのレジスタを2本使用します．つまり2本のレジスタでセグメントアドレスとオフセットアドレスを指定するのです．

セグメントアドレスを指定するために使われるのがセグメントレジスタです．オフセットアドレスを指定するレジスタに関しては次節の「プログラム実行のメカニズム」で取り上げることにします．

メモリをアクセスするには，あらかじめ16ビットのセグメントアドレスの値をセグメントレジスタにセットしておいて，16ビットのオフセットアドレスでアドレスを指定するという手順をとります．セグメントレジスタは全部で4つありますが，各セグメントレジスタの役割を次の**表5-2**に示しておきます．

セグメントレジスタ	名称	機能	用途
CS	コードセグメント	マシン語プログラムが格納されているセグメントのセグメントアドレスを指定する	CPUが実行するマシン語命令を読み込む際に自動的に使用される
DS	データセグメント	データを格納するセグメントのセグメントアドレスを指定する	メモリとレジスタの間でデータの転送を行う際に自動的に使用される
ES	エクストラセグメント	DSで指定するセグメント以外にデータを格納するセグメントが必要な場合に指定する	メモリとレジスタの間でデータの転送を行う際に使用される
SS	スタックセグメント	スタック専用に使うセグメントのセグメントアドレスを指定する	スタックを操作する際に自動的に使用される

表5-2　セグメントレジスタとその役割

この表の「用途」の項目で示したように，メモリからマシン語命令を読み込んだり，メモリにデータを転送したりする際に，各セグメントレジスタが用途に応じて自動的に選択され，セグメントベースとして設定されます．つまり，一度セグメントレジスタにセグメントアドレスをセットしておけば，オフセットアドレスだけをアドレスとして考えればよいのです．

MS-DOSでは，プログラムを起動する際にメモリ空間にセグメントを割り当てて，自動的にそのセグメントアドレスをセグメントレジスタにセットし

てくれます．たとえば，かな漢字変換システムやRAMディスクドライバなどを組み込むと，それぞれにセグメントが割り当てられ，その後に割り当てられるプログラムのセグメントのアドレスが変化するのです．ですから本書のリストにおけるセグメントアドレス（□□□□：○○○○の□□□□の部分およびセグメントレジスタの内容）は，みなさんが自分で実際に試してみる場合とはおそらく異なっているでしょう．このアドレスはMS-DOSが自動的に割り当ててくれるものなので，異なっていても心配する必要はありません．そのまま実習を行ってください．

なお，本書ではすべてのセグメントレジスタの内容が同じ場合，すなわち1つのセグメントのみを扱うことにします．以下に，本書で利用するセグメントの配置を示します．

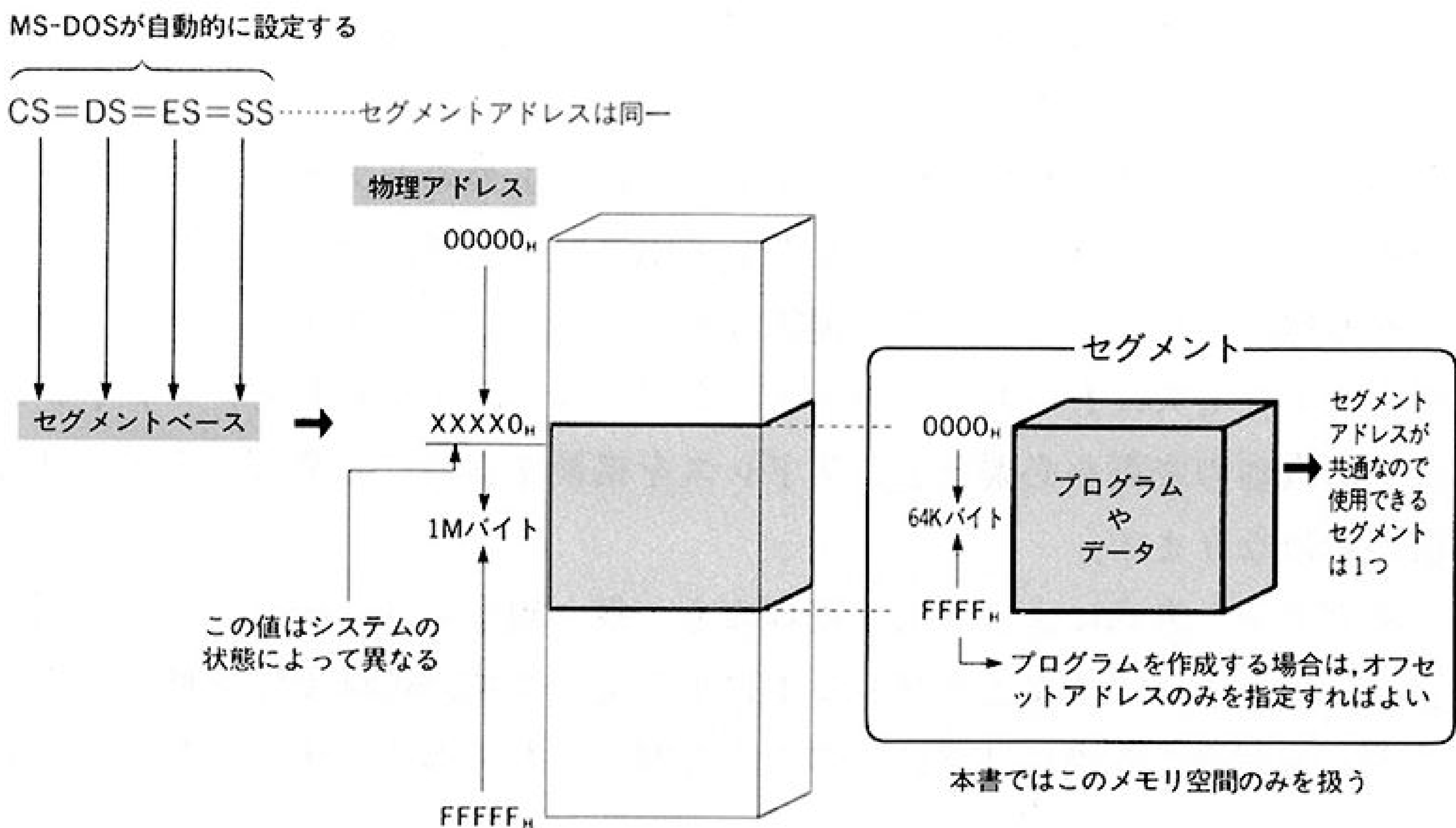


図5-14　本書で使用するセグメント

セグメント方式の利点と欠点

最後に，セグメント方式の利点と欠点をまとめておきます．

16 ビット CPU には，68000CPU のように 1 本のレジスタで物理アドレスを直接指定できるものも少なくありません．8086CPU はなぜこのようにめんどうな 2 段階のアドレス指定方式を採用しているのでしょうか．

多くの場合，64K バイト以上のデータをアクセスしたり 64K バイトのプログラムが必要になることはあまりありません．少なくともプログラムやデータをあるブロックで区切ってしまえば，たいていの場合それぞれのブロックは 64K バイト以内に収まってしまいます．

あるセグメント内のデータを処理している間や，あるセグメント内のプログラムを実行している間は，セグメントアドレスは固定でありオフセットアドレスだけを扱えばよいので，CPU はアドレス計算を高速に行うことができ，アドレスを格納するのに必要なメモリも少なくてすみます．8 ビット CPU のようなコンパクトさを保ちながら大きなメモリ空間を扱うことができるのがセグメント方式の特徴です．

しかしこの方法にも欠点があります．それは 1 つのセグメントの大きさが 64K バイトまでに制限されることです．64K バイト以上にわたる連続したデータを処理しようとすると，この制限は大きな障害になります．1 バイトのメモリのアクセスにもいちいちセグメントアドレス，すなわちセグメントレジスタの内容の変更を必要とし，アドレスを格納するのに必要とするメモリも倍の量になります．

セグメント方式によるメモリ管理にも一長一短があり，セグメント方式によるメリットはそのままデメリットにもなるのです．8086CPU を使ってプログラムを開発する際には常にこのことを頭に入れておく必要があります．

5.6 プログラム実行のメカニズム

●

8086CPUの基本的な概念は，これまでの解説でおおよそつかめたでしょうか．この節では，これまでの基礎事項を踏まえた上で，8086CPUがプログラムを実行するメカニズムを追ってみましょう．これを理解することは，すなわち一般的なコンピュータの動作原理を理解することでもあります．

ここでは，実際にDEBUGを使っていきながら，プログラムの実行手順を見ていくことにしましょう．

現在使われているコンピュータのほとんどは，コンピュータの生みの親と呼ばれるフォン・ノイマンの提案した動作原理を基に設計されています*．その原理は，プログラムはメモリに格納され，CPUがそれを順次読み出して解析，実行するというプログラム記憶方式（ストアドプログラムシーケンシャルコントロール方式）にあります．この方式は現在のコンピュータの仕組みの大原則であり，8086CPUも例外ではありません．

CPUの実行するマシン語はメモリに格納され，同じくメモリに格納される単なるデータと見かけ上は区別がつきません．2章で実行型ファイルをダンプした時も，マシン語のプログラムにはマシン語の命令と文字列などのデータが混在していました．

文字列などのデータを誤ったプログラムなどによりマシン語命令として解釈した場合は，でたらめなマシン語命令の組合せとして実行されてしまいます．これではプログラムの実行が暴走することになり，多くの場合たいへん危険なことです．プログラムとデータを区別し管理するのは，プログラムを作成する人の務めです．プログラムを作るときには，データを誤ってプログラムとして実行したり，プログラム領域を誤ってデータ領域として扱いプロ

*ノイマン型アーキテクチャはすでに限界に達しているとして，さらに効率のよい非ノイマン型の動作原理がいろいろ模索されている．

グラムを破壊したりすることのないように注意しなければなりません．

プログラムの作成

「5.2 ニーモニックとアセンブラ」で使ったマシン語のプログラムを再び例として取り上げましょう．これは93ページに示した**図5-4**のようにDEBUGのAコマンドを使ってアセンブリ言語のニーモニックで入力したものです．実習を行うには**図5-4**を参照してプログラムを入力してください．なお，Q (Quit) コマンドでDEBUGを終了してしまった場合は，メモリにセットしたプログラムやデータは失われてしまうので注意してください．

マシン語とニーモニックの対応を見るために，もう一度Uコマンドで逆アセンブルしてみます（**図5-15**）．

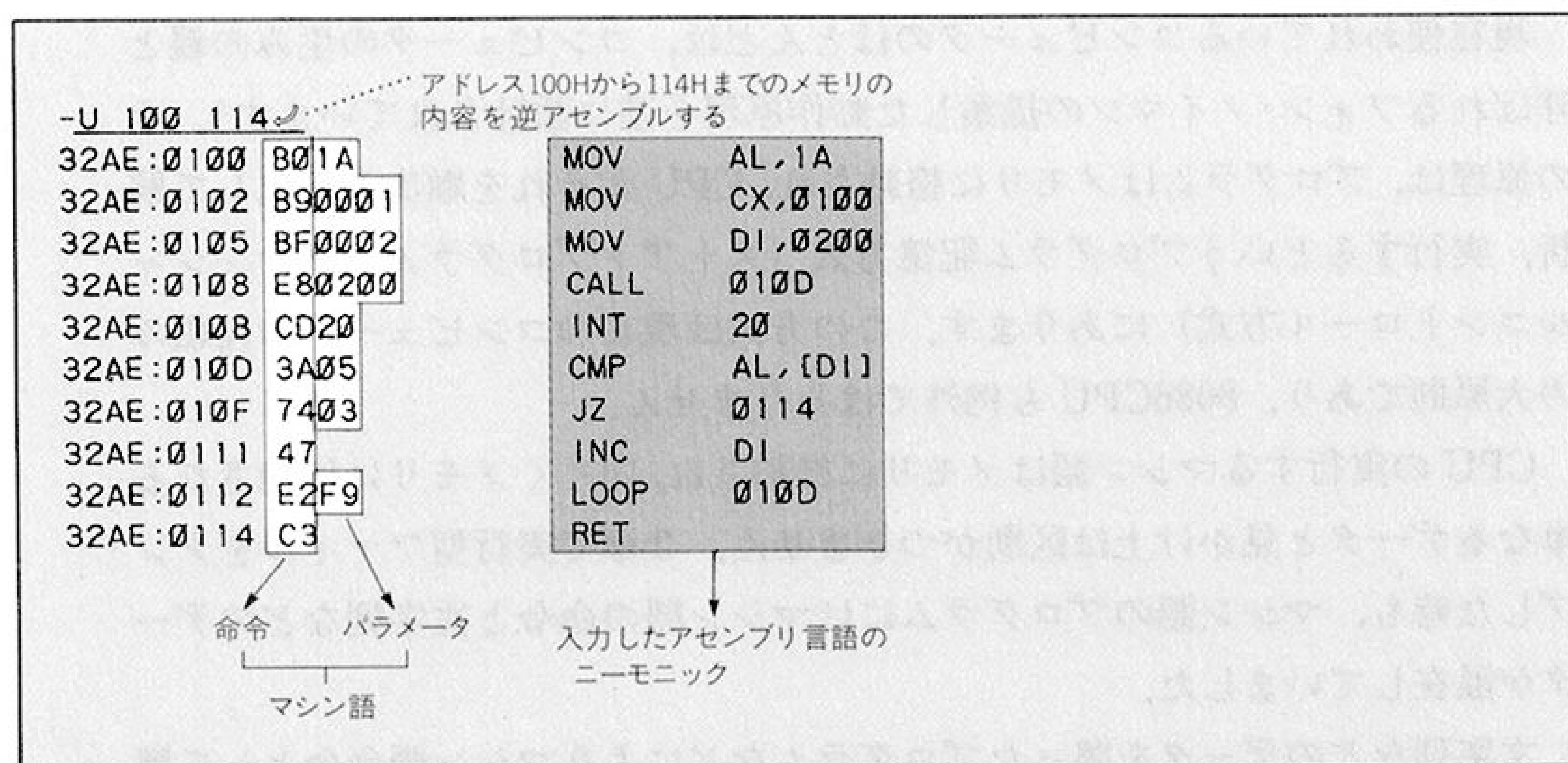


図5-15　5.2節で作成したプログラム

この図を見ればわかるように，マシン語の1命令は，「命令の種類」を表すコードとその「パラメータ」となるデータから成っています．図の中で色の付いている部分が命令の種類を表すコードであり，その他がパラメータです．

実はこのプログラムは，現在のデータセグメントのオフセットアドレス200_Hから始まる100_Hバイトの間に値「$1A_H$」* があるかどうかを調べる，というも

のです．このプログラムがその値を見つけられることを確かめるために，オフセットアドレス201_Hに$1A_H$を書き込んでおきましょう．メモリの値を変更するには，3章でメッセージを変更するのに使用したE (Enter) コマンドを使います（図5-16）．

```
-D 200 201↵ ……………データを変更する前にメモリの内容を確認する

32AE:0200  FE 29
  ┌─E(Enter)コマンド……………メモリの内容を変更する
-E 200 31,1A↵ ……………アドレス200Hと201Hのメモリの内容をそれぞれ
先頭アドレス─┘   └──変更するメモリの値        「31H」,「1AH」に変更する
-D 200 201↵ ……………正しく書き込まれたかどうかD(Dump)コマンド
                         で調べる

32AE:0200  31 1A
-
```

図5-16　オフセットアドレス201_Hに$1A_H$を書き込む

●

Tコマンドによるプログラムのトレース

まず，Rコマンドでレジスタの値を調べておきましょう．すべてのセグメントレジスタが同じ値にセットされていることに注目してください．

```
-R↵ ……………現在のレジスタの値と次に実行される命令が表示される
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=32AE─ES=32AE─SS=32AE─CS=32AE─ IP=0100   NV UP DI PL NZ NA PO NC
32AE:0100 B01A          MOV   AL,1A
-     ↓        ↓              ↓                ＼セグメントレジスタの値はすべて一致
                                                 (ただし，この値はシステムによって異なる)
次に実行され  次に実行され   次に実行される命令
るプログラムの るマシン語     のニーモニック
アドレス
```

図5-17　実行前の各レジスタの値

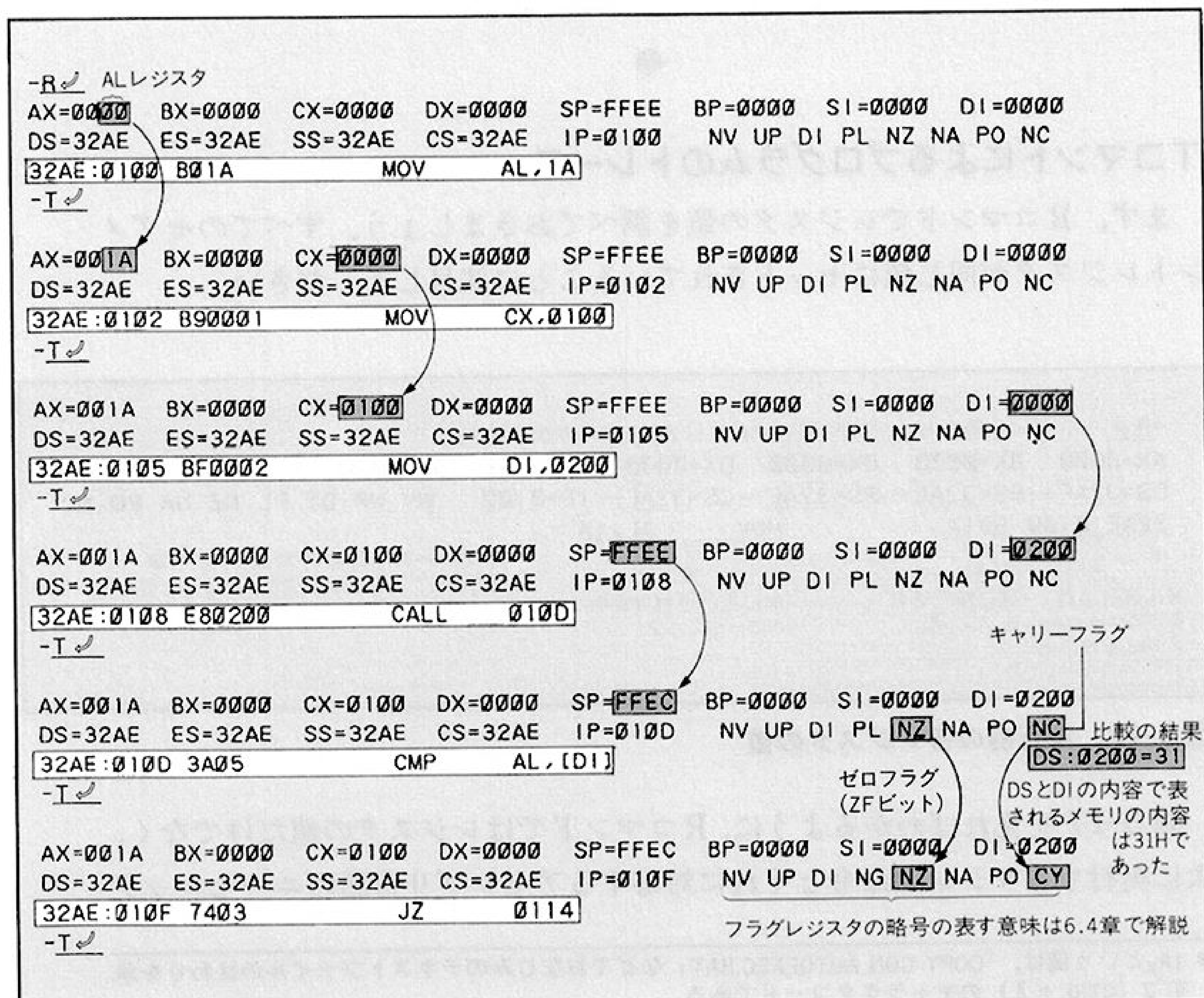
図5-17を見ればわかるように，Rコマンドではレジスタの値だけでなく，次に実行するマシン語命令とそれに対応するアセンブリ言語のニーモニック

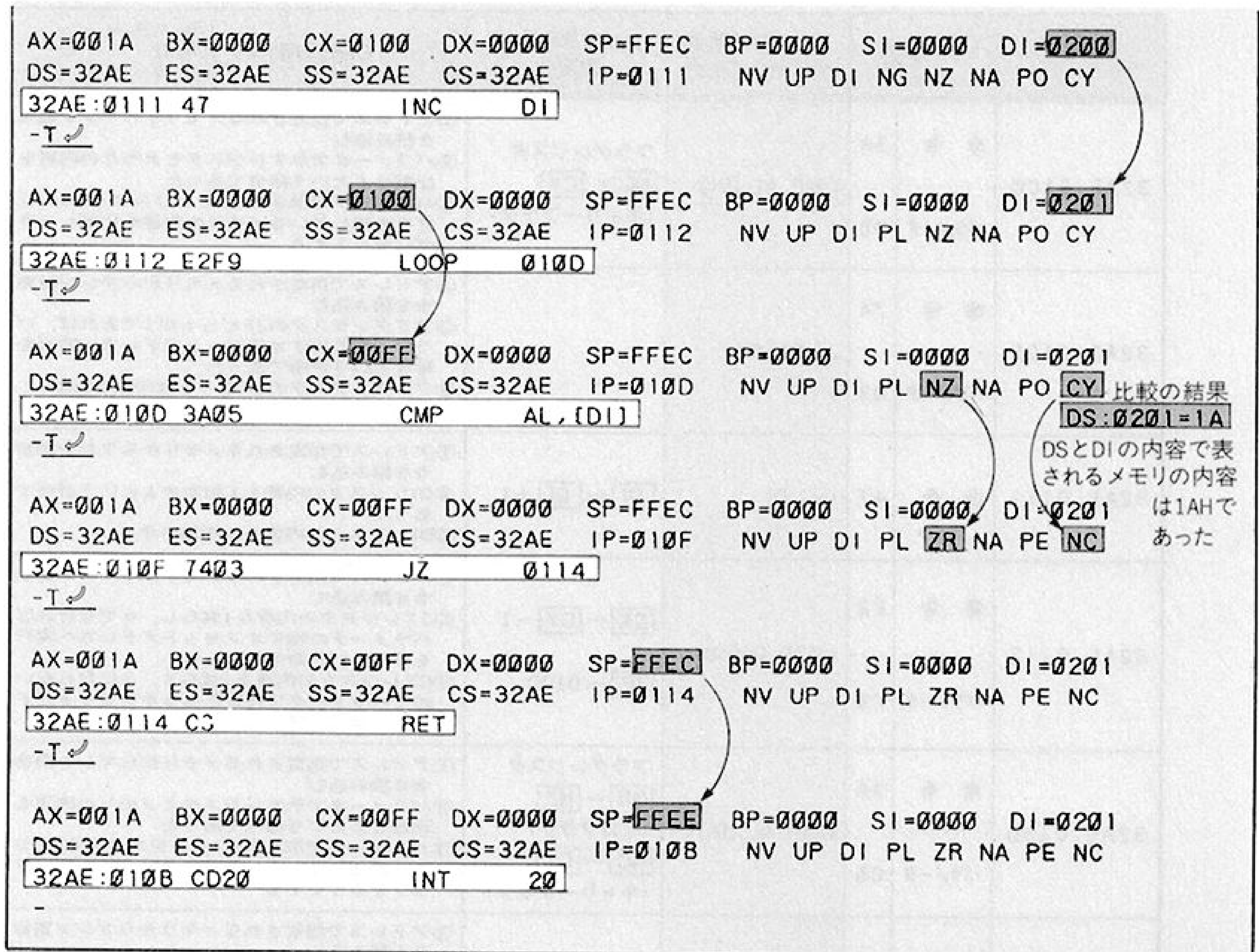
＊ $1A_H$という値は，「COPY CON AUTOEXEC.BAT」などでおなじみのテキストファイルの終わりを示す^Z（CTRL＋Z）のキャラクタコードである．

が表示されます．このためプログラムを実行する前にRコマンドを入力することにより，最初に実行するマシン語命令を確認することができます．

それではプログラムを実行してみましょう．プログラムの動作やそれにともなうレジスタの値の変化を確かめるために，1命令ずつ実行するT（Trace）コマンドを使います．Tコマンドは1命令実行すると，Rコマンドのように各レジスタの値を表示して，停止します．このときもマシン語命令とそれに対応するアセンブラのニーモニックが表示されますが，これは今，実行した命令ではなく次に実行される命令であることに注意してください．

プログラムがどのように実行されているのかを，次の図 **5-18** でTコマンドの実行結果にしたがって解説しましょう．ここでは1つ1つの命令の意味を理解する必要はありませんが，レジスタへデータを読み込み，演算を行っているという様子を把握してください．





```
AX=ØØ1A  BX=ØØØØ  CX=Ø1ØØ  DX=ØØØØ  SP=FFEC  BP=ØØØØ  SI=ØØØØ  DI=Ø2ØØ
DS=32AE  ES=32AE  SS=32AE  CS=32AE  IP=Ø111   NV UP DI NG NZ NA PO CY
32AE:Ø111 47            INC     DI
-T
AX=ØØ1A  BX=ØØØØ  CX=Ø1ØØ  DX=ØØØØ  SP=FFEC  BP=ØØØØ  SI=ØØØØ  DI=Ø2Ø1
DS=32AE  ES=32AE  SS=32AE  CS=32AE  IP=Ø112   NV UP DI PL NZ NA PO CY
32AE:Ø112 E2F9          LOOP    Ø1ØD
-T
AX=ØØ1A  BX=ØØØØ  CX=ØØFF  DX=ØØØØ  SP=FFEC  BP=ØØØØ  SI=ØØØØ  DI=Ø2Ø1
DS=32AE  ES=32AE  SS=32AE  CS=32AE  IP=Ø1ØD   NV UP DI PL NZ NA PO CY
32AE:Ø1ØD 3AØ5          CMP     AL,[DI]
-T
AX=ØØ1A  BX=ØØØØ  CX=ØØFF  DX=ØØØØ  SP=FFEC  BP=ØØØØ  SI=ØØØØ  DI=Ø2Ø1
DS=32AE  ES=32AE  SS=32AE  CS=32AE  IP=Ø1ØF   NV UP DI PL ZR NA PE NC
32AE:Ø1ØF 74Ø3          JZ      Ø114
-T
AX=ØØ1A  BX=ØØØØ  CX=ØØFF  DX=ØØØØ  SP=FFEC  BP=ØØØØ  SI=ØØØØ  DI=Ø2Ø1
DS=32AE  ES=32AE  SS=32AE  CS=32AE  IP=Ø114   NV UP DI PL ZR NA PE NC
32AE:Ø114 C3            RET
-T
AX=ØØ1A  BX=ØØØØ  CX=ØØFF  DX=ØØØØ  SP=FFEE  BP=ØØØØ  SI=ØØØØ  DI=Ø2Ø1
DS=32AE  ES=32AE  SS=32AE  CS=32AE  IP=Ø1ØB   NV UP DI PL ZR NA PE NC
32AE:Ø1ØB CD2Ø          INT     2Ø
-
```


図5-18　プログラムの実行手順(1)

アドレス	マシン語		アセンブリ言語のニーモニック	レジスタの内容の変化	CPUの動作　①読み込み ②解読 ③実行
32AE:0100	命　令	B0	MOV AL,1AH	AL ← $1A_H$	①アドレスで指定されるメモリからマシン語命令を読み込む ②ALレジスタにパラメータ(1バイト)を読み込めという命令であった ③パラメータ$1A_H$をALレジスタに読み込む
	パラメータ	1A			
32AE:0102	命　令	B9	MOV CX,0100H	CX ← 0100_H	①アドレスで指定されるメモリからマシン語命令を読み込む ②CXレジスタにパラメータ(1ワード)を読み込めという命令であった ③パラメータ0100_HをCXレジスタに読み込む (マシン語では00,01と上位／下位がひっくり返っていることに注意)
	パラメータ	00 01			
32AE:0104	命　令	BF	MOV DI,0200H	DI ← 0200_H	①アドレスで指定されるメモリからマシン語命令を読み込む ②DIレジスタにパラメータ(1ワード)を読み込めという命令であった ③パラメータ0200_HをDIレジスタに読み込む
	パラメータ	00 02			
32AE:0108	命　令	E8	CALL 010DH	SP ← SP −2 IP ← $010D_H$	①アドレスで指定されるメモリからマシン語命令を読み込む ②パラメータの指すオフセットアドレスのサブルーチンを実行せよという命令であった (パラメータに対応するマシン語は相対アドレスを表すのでオフセットアドレスそのものではない) ③次の命令のオフセットアドレス(010B)をスタックに積みパラメータの指すオフセットアドレスに実行を移す
	パラメータ	02 00			

(次ページへ続く)

アドレス	マシン語		アセンブリ言語のニーモニック	レジスタの内容の変化	CPUの動作 ①読み込み ②解説 ③実行
32AE:010D	命 令 パラメータ	3A 05	CMP AL,[DI]	フラグレジスタ NC ← CY (キャリーフラグ)	①アドレスで指定されるメモリからマシン語命令を読み込む ②パラメータで示すレジスタとメモリの内容を比較せよという命令であった ③パラメータで示されるALレジスタとDIレジスタの指しているメモリの内容を比較しフラグにセットする
32AE:010F	命 令 パラメータ	74 03	JZ 0114H		①アドレスで指定されるメモリからマシン語命令を読み込む ②フラグレジスタのZFビットが1であれば，パラメータの指すオフセットアドレスへ実行を移せという命令であった ③フラグレジスタのZFビットは0なので何もしない
32AE:0111	命 令	47	INC DI	DI ← DI +1	①アドレスで指定されるメモリからマシン語命令を読み込む ②DIレジスタの内容を1増加せよという命令であった ③DIレジスタの内容を1増加させる
32AE:0112	命 令 パラメータ	E2 F9	LOOP 010DH	CX ← CX −1 IP ← $010D_H$	①アドレスで指定されるメモリからマシン語命令を読み込む ②CXレジスタの内容を1減らし，0でなければパラメータの指すオフセットアドレスへ実行を移せという命令であった ③CXレジスタの内容を1減らす，0にならないのでパラメータで指定されるオフセットアドレスへ実行を移す
32AE:010D	命 令 パラメータ	3A 05	CMP AL,[DI]	フラグレジスタ NR ← NZ (ゼロフラグ) NC ← CY (キャリーフラグ)	①アドレスで指定されるメモリからマシン語命令を読み込む ②パラメータで示すレジスタとメモリの内容を比較せよという命令であった ③パラメータで示されるALレジスタとDIレジスタの指しているメモリの内容を比較し，フラグをセットする
32AE:010F	命 令 パラメータ	74 03	JZ 0114H	IP ← 0114_H	①アドレスで指定されるメモリからマシン語命令を読み込む ②フラグレジスタのZFビットが1であれば，パラメータの指すオフセットアドレスへ実行を移せという命令であった ③フラグレジスタのZFビットが1なのでパラメータで指定されるオフセットアドレスへ実行を移す
32AE:0114	命 令	03	RET	IP ← $010B_H$ SP ← SP +2	①アドレスで指定されるメモリからマシン語命令を読み込む ②サブルーチンからメインルーチンへ復帰せよという命令であった ③スタックからオフセットアドレスを取り出し，そこへ実行を移す
32AE:010B	命 令 パラメータ	CD 20	INT 20		①アドレスで指定されるメモリからマシン語命令を読み込む ②パラメータで指定される内部割り込みを発生せよという命令であった ③パラメータ20_Hの内部割り込みを発生する．(その結果，プログラムを終了し，DEBUGに戻る)

図 5-18　プログラムの実行手順(2)

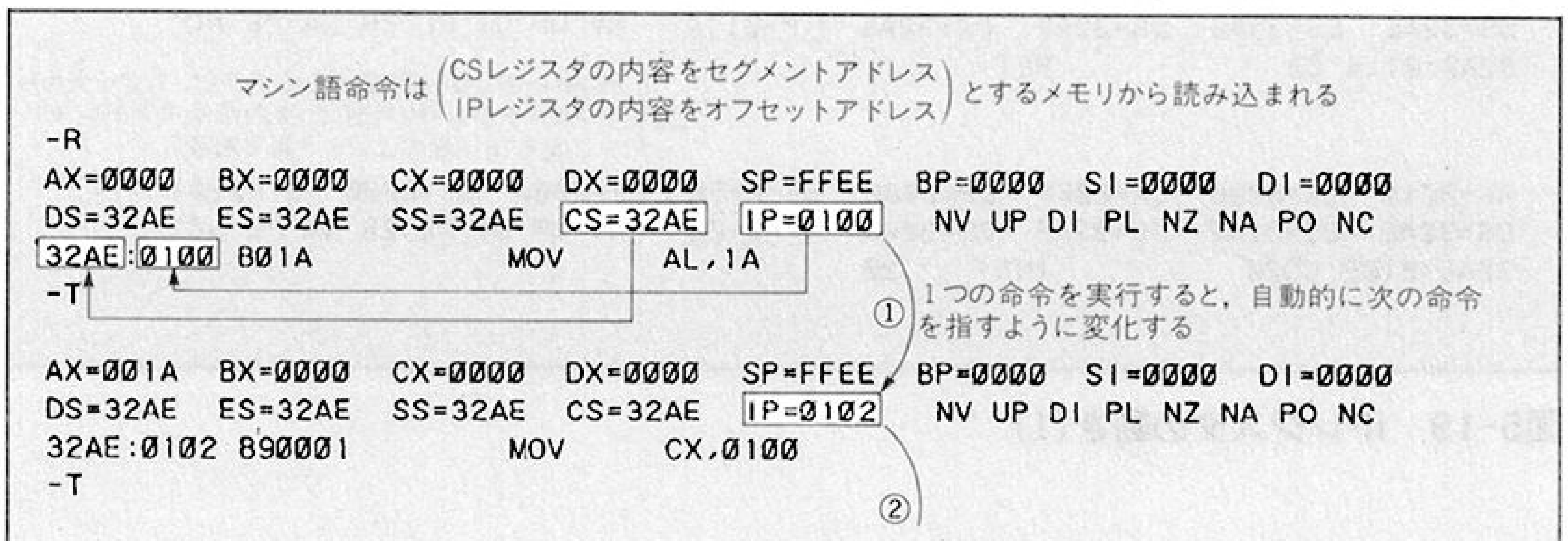
IP レジスタの役割

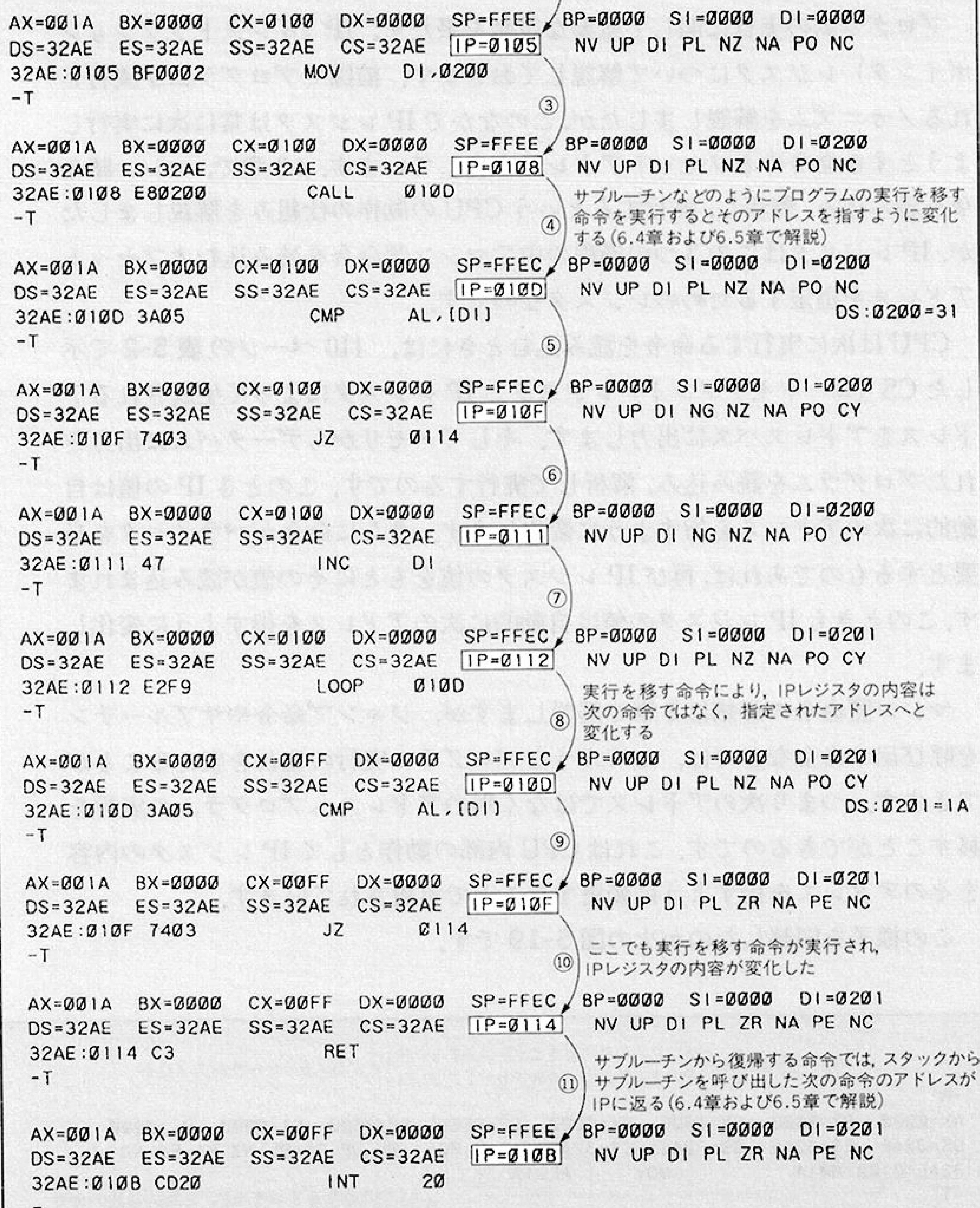
プログラムの実行に関して重要な役割を果たす，IP（インストラクションポインタ）レジスタについて解説しておきます．前図でプログラムが実行されるメカニズムを解説しましたが，このなかでIPレジスタは常に次に実行しようとする命令のオフセットアドレスを指しています．4.3章で，マシン語命令を読み込み，解釈し，実行するというCPUの動作の仕組みを解説しましたが，IPレジスタはこの3つの過程の中でマシン語命令を読み込むオフセットアドレスを指定するためのレジスタなのです．

CPUは次に実行する命令を読み込むときには，110ページの**表5-2**で示したCS（コードセグメント）レジスタとIPレジスタによって生成されるアドレスをアドレスバスに出力します．そしてメモリからデータバスに出力されたプログラムを読み込み，解析して実行するのです．このときIPの値は自動的に次のアドレスを指すように変化します．さらに命令がパラメータを必要とするものであれば，再びIPレジスタの値をもとにその値が読み込まれます．このときもIPレジスタの値は自動的に次のアドレスを指すように変化します．

マシン語命令の詳細は6章で解説しますが，ジャンプ命令やサブルーチンを呼び出す命令などでは，このようなプログラム実行の流れを変えることができます．つまり次のアドレスではなく別のアドレスへプログラムの実行を移すことができるのです．これはCPU内部の動作としてIPレジスタの内容をそのアドレスを指すように変更することで実現されています．

この様子を図解したのが次の**図5-19**です．





```
AX=001A  BX=0000  CX=0100  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=32AE  ES=32AE  SS=32AE  CS=32AE  IP=0105   NV UP DI PL NZ NA PO NC
32AE:0105 BF0002        MOV     DI,0200
-T

AX=001A  BX=0000  CX=0100  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0200
DS=32AE  ES=32AE  SS=32AE  CS=32AE  IP=0108   NV UP DI PL NZ NA PO NC
32AE:0108 E80200        CALL    010D
-T

AX=001A  BX=0000  CX=0100  DX=0000  SP=FFEC  BP=0000  SI=0000  DI=0200
DS=32AE  ES=32AE  SS=32AE  CS=32AE  IP=010D   NV UP DI PL NZ NA PO NC
32AE:010D 3A05          CMP     AL,[DI]                        DS:0200=31
-T

AX=001A  BX=0000  CX=0100  DX=0000  SP=FFEC  BP=0000  SI=0000  DI=0200
DS=32AE  ES=32AE  SS=32AE  CS=32AE  IP=010F   NV UP DI NG NZ NA PO CY
32AE:010F 7403          JZ      0114
-T

AX=001A  BX=0000  CX=0100  DX=0000  SP=FFEC  BP=0000  SI=0000  DI=0200
DS=32AE  ES=32AE  SS=32AE  CS=32AE  IP=0111   NV UP DI NG NZ NA PO CY
32AE:0111 47            INC     DI
-T

AX=001A  BX=0000  CX=0100  DX=0000  SP=FFEC  BP=0000  SI=0000  DI=0201
DS=32AE  ES=32AE  SS=32AE  CS=32AE  IP=0112   NV UP DI PL NZ NA PO CY
32AE:0112 E2F9          LOOP    010D
-T

AX=001A  BX=0000  CX=00FF  DX=0000  SP=FFEC  BP=0000  SI=0000  DI=0201
DS=32AE  ES=32AE  SS=32AE  CS=32AE  IP=010D   NV UP DI PL NZ NA PO CY
32AE:010D 3A05          CMP     AL,[DI]                        DS:0201=1A
-T

AX=001A  BX=0000  CX=00FF  DX=0000  SP=FFEC  BP=0000  SI=0000  DI=0201
DS=32AE  ES=32AE  SS=32AE  CS=32AE  IP=010F   NV UP DI PL ZR NA PE NC
32AE:010F 7403          JZ      0114
-T

AX=001A  BX=0000  CX=00FF  DX=0000  SP=FFEC  BP=0000  SI=0000  DI=0201
DS=32AE  ES=32AE  SS=32AE  CS=32AE  IP=0114   NV UP DI PL ZR NA PE NC
32AE:0114 C3            RET
-T

AX=001A  BX=0000  CX=00FF  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0201
DS=32AE  ES=32AE  SS=32AE  CS=32AE  IP=010B   NV UP DI PL ZR NA PE NC
32AE:010B CD20          INT     20
-
```


図5-19　IPレジスタの動き(1)

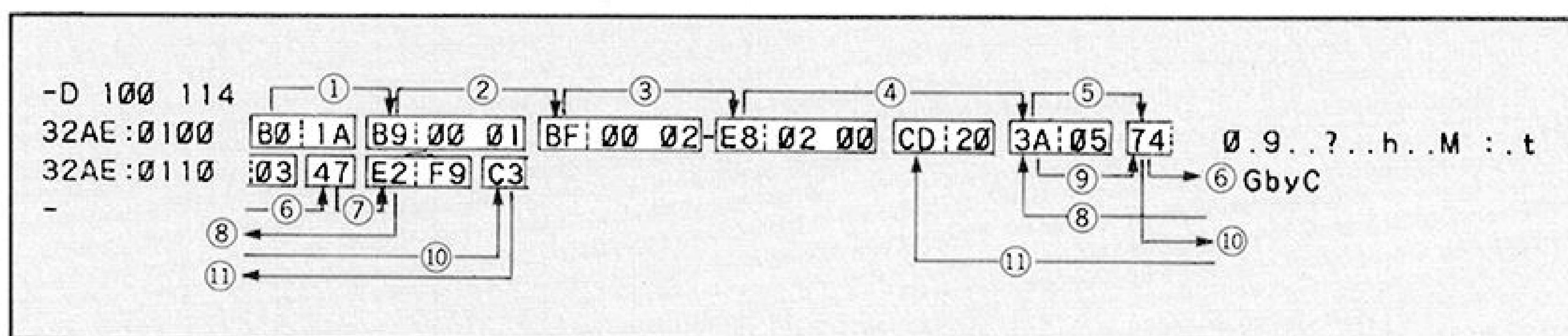
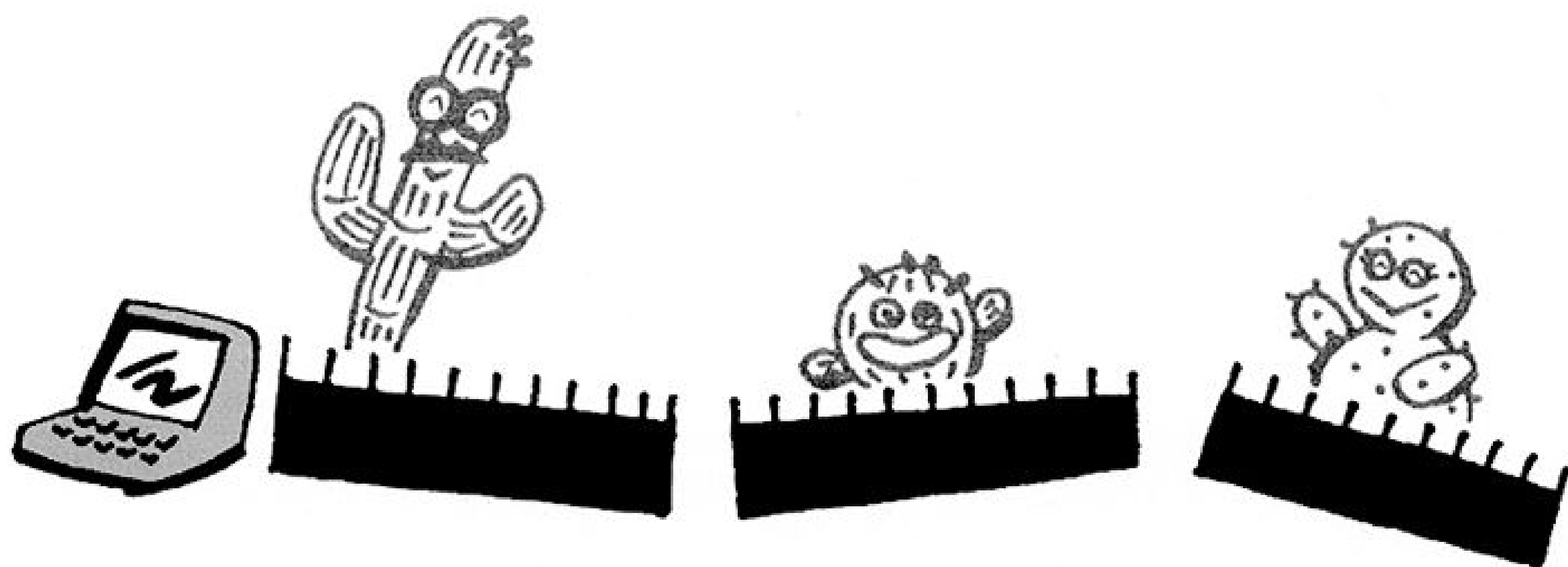


図5-19　IPレジスタの動き(2)

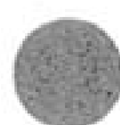
この例にも登場していますが，演算の結果などがある条件を満たしているときにだけ実行の流れを変えるマシン語命令も用意されています．このようにプログラム実行の流れを条件付きで変えることによって，「演算」とともにコンピュータの基本的な機能である「判断」を実現するのです．

5.3節でマシン語のプログラムとはレジスタを使うことであると解説しました．マシン語のプログラムを学習するためのもう1つの重要なポイントは，プログラム実行の流れを制御する点にあります．



6

マシン語命令の実習



コンピュータ・システムの仕組みや，CPU がプログラムを実行する仕組みはほぼ理解できたと思いますが，本章からはマシン語そのものの実習に入ります．実習では DEBUG を使ってプログラムをメモリに書き込み，それを実行して結果を確かめながら各命令の働きを解説していきます．

マシン語の１つ１つの命令はどれも単純で明解です．この単純明解な命令を組み合わせていくことにより，非常に高度なソフトウェアをプログラミングすることも可能になるのです．目的とするプログラムを作成するために，マシン語命令をどのように組み合わせていけばよいかは「プログラミングテクニック」の問題であり，本書を卒業した次の段階では重要なこととなります．

本章では，マシン語の基礎である個々の命令について解説しますが，マシン語の応用について多くを取り上げているわけではありません．まず個々の命令を学習することにより，CPU の「仕組みと働き」（アーキテクチャ）を理解することが最も大切なのです．

6.1　実習の前に

8086CPUは約100種類のインストラクション・セット（命令群）を持っており，すべての命令の組合せを数えると膨大な数になります．しかし，ほとんどのプログラムはせいぜい数十種類の命令だけでできています．どんなに大規模なプログラムでも，全命令の1～2割程度しか使われていないと考えてよいでしょう．

ですから最初からたくさんの命令を覚える必要はありません．20～30の命令を組み合わせれば，たいていのプログラムは不自由なく組むことが可能になります．それ以外の命令は必要に応じて覚えていけばよいものです．

そして，これは重要なことですが，マシン語のプログラムは工夫次第で短縮できたり，また実行速度が速いものを作ることができます．しかし，そのようなテクニックを駆使することだけがマシン語のプログラミングではありません．確かにプログラムによっては高速化のための技法を使う必要も出てくるでしょう．だからといって最初からテクニックに凝る必要はまったくありません．基本命令を使って，「きちんと整理された，わかりやすいプログラムを組むこと」が第一なのです．

インストラクションのすべての組合せを機能別のグループに分け，わかりやすく整理した一覧表が，各CPUの「インストラクション・セット一覧表」と言われるもので，各CPUの製造会社から発売されています．みなさんがアセンブラを使い始めると，座右の表として参照することになるでしょう．

しかし，しだいに経験を積んでプログラミングの力が備わってくると，よほどの特殊な目的でない限り，この表を見ない人が多くなるようです．これは，人によって多少の差はありますが，通常のプログラムを組む場合に使う命令の種類がだいたい固定されてくるからなのです．自分にとって使い慣れたなじみの命令や，覚えやすい命令をどうしても多く使うようになるのです．またほとんどのプログラムは，これらのよく知っている命令だけで十分作成

することができます．

本章で大切なことは，8086CPU独自の特別な機能を追求するより，コンピュータの基本動作に関わる一般的な基本命令を正しく理解することです．1つ1つのマシン語そのものよりも，コンピュータの基本的なアーキテクチャを理解してほしいのです．そうすれば，自在にプログラムが組めるようになるだけでなく，コンピュータ全般についても理解が深まるでしょう．

なお，本章では実習プログラムの入力や実行を，これまでと同様にDEBUGコマンドを使って行います．DEBUGコマンドの基本的な使い方については何度か実習してきましたが，以下にもう一度，表を使って整理しておくことにします．

コマンド名	書　式	機　能
アセンブル Assemble	A［〈アドレス〉］	入力されたニーモニックをアセンブルしてマシン語に変換し，指定した〈アドレス〉以降のメモリに収める
ダンプ Dump	D［〈アドレス1〉］［〈アドレス2〉］	〈アドレス1〉から〈アドレス2〉までのメモリの内容を16進ダンプとキャラクタコードで表示する
エンター Enter	E〈アドレス〉［〈リスト〉］	〈アドレス〉から始まるメモリの内容をリストで指定する値に変換する
ゴー Go*	G［=〈アドレス1〉］［〈アドレス2〉］	〈アドレス1〉から〈アドレス2〉までのマシン語プログラムを実行する
ネーム Name*	N〈ファイル名〉	マシン語プログラムのファイル名を指定する
クイット Quit	Q	DEBUG(SYMDEB)を終了し，MS-DOSに戻る
レジスタ Register	R［〈レジスタ名〉］	全レジスタの内容とIPレジスタの指し示す命令を表示する．レジスタ名を指定すると，そのレジスタの内容を変更できる
トレース Trace	T［=〈アドレス〉］［命令数］	〈アドレス〉で示されるメモリのプログラムを1命令実行し，全レジスタの内容と次に実行する命令を表示する
アセンブル Unassemble	U［〈アドレス1〉］［〈アドレス2〉］	〈アドレス1〉から〈アドレス2〉までのメモリの内容を逆アセンブルして表示する
ライト Write	W	マシン語プログラムをBXレジスタとCXレジスタで設定したバイト数だけディスクにファイルとして書き出す

〈この表の見方〉・〈アドレス〉は，「○○○○」という16進数で4桁のオフセットアドレスを指定する．
・〈リスト〉は，「○○，○○，……」というように16進数の2桁をカンマで区切って指定する．
・［　］は，その項目が省略可能であることを意味する．

〈注意〉
・書式は，本章の実習で行うもののみを選択してある．くわしくは，APPENDIXの「DEBUG(SYMDEB)主要コマンド一覧」を参照のこと．
・＊マークの付いたコマンドは，本章で初めて実習するコマンドである．

表6-1　実習で使用するDEBUG（SYMDEB）のコマンド一覧

6.2　データ転送命令

データ転送命令とは，8ビットまたは16ビットのデータをCPUのレジスタどうし，またはレジスタとメモリの間でやりとりする命令のことです．

アセンブリ言語のニーモニックでは，データ転送命令は次のような形式で表します．

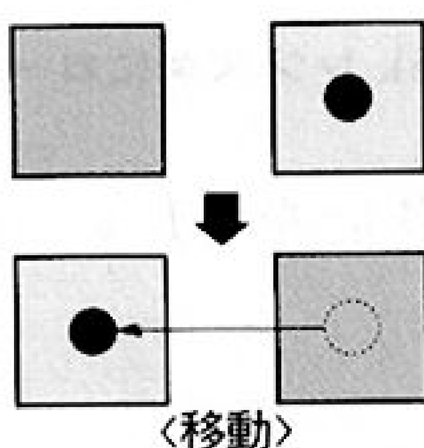
MOV（ムーブ）(MOVe)　**受け取る側　，　送り出す側**

（レジスタまたは メモリ） ← **（レジスタまたは メモリ，数値）**

このようにデータ転送命令は，右側に指定したレジスタやメモリのデータ格納場所から(あるいはデータそのものを)，左側に指定したデータ格納場所へ「転送する」という命令です．データの移動する向きは必ず「右から左」ということをよく覚えておいてください．この点は，COPYなどのMS-DOSのコマンドとは方向が逆になるので注意が必要です．

また，転送というと誤解を招きやすいのですが，実際には右側に指定されたデータの格納場所からデータが移動してしまって，そこからデータがなくなるわけではありません．**図6-1**のように転送元のデータはそのままで，転送先に同じ値のデータを受け渡すということです．どちらかといえば，「転送」というよりは「複写」というイメージが近いでしょう．

MOVE（ムーブ）というともとのデータが消えてしまいそうだが……



実際にはデータがCOPY（コピー）されるので，もとのデータはそのまま残る

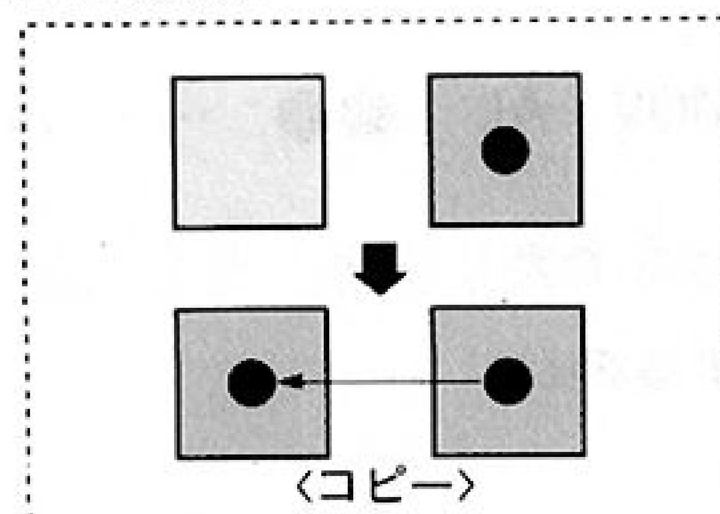


図6-1　データ転送の概念図

メモリとCPUのレジスタ間で行うデータのやり取りに関しては，言葉の上では習慣として，

レジスタ　←　メモリ　　の場合を「ロード」
レジスタ　→　メモリ　　の場合を「ストア」

と呼んでいます．しかしマシン語の命令，すなわちアセンブリ言語のニーモニックの上では，どちらの場合もすべて「MOV」だけなので注意してください．

以下に，一般的に使われる「ロード」，「ストア」，「セーブ」，「転送」の意味を図解しておきましょう．

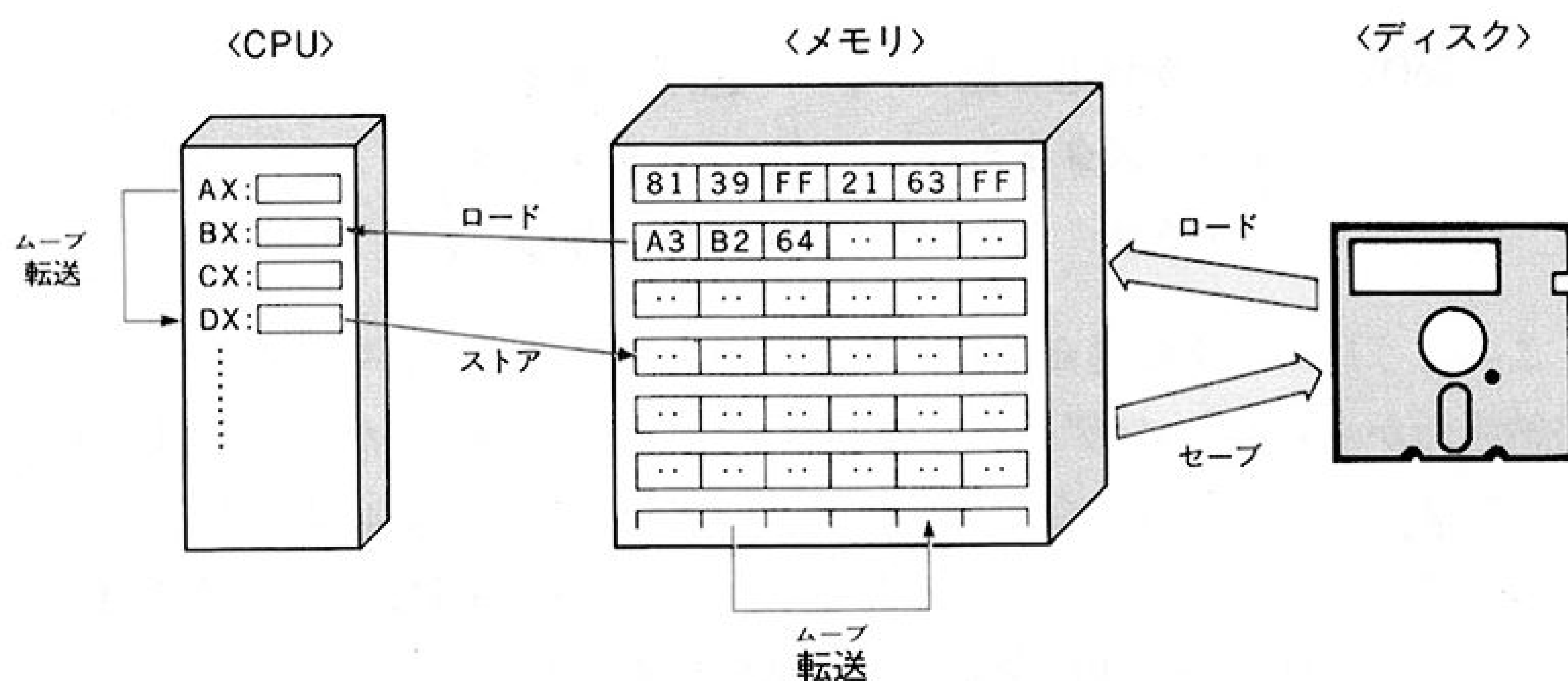


図 6-2　「ロード」，「ストア」，「セーブ」，「転送」の意味

実習 1　1 バイトデータをメモリに書き込む

任意の1バイトのデータ（00_H～FF_H）をALレジスタにロードし，そのまま任意のアドレスのメモリにストアしてみましょう．

「ALレジスタ←1バイトのデータ」の命令は，

MOV　AL，●●　……1バイトデータ「●●」をALレジスタにロードする．

という形式で表します．たとえば1バイトのデータ「FF_H」をALレジスタにロードするのは，

MOV　AL , 0FFH …… 「FF_H」は16進数であることを明確にするために先頭に0を付ける*.

となります.

次に「メモリ← ALレジスタの内容」の命令は,

MOV　[○○●●], AL …… ALレジスタの内容をオフセットアドレス「○○●●」で指し示すメモリにストアする.

という形式で表します. たとえばALレジスタの内容をオフセットアドレス「201_H」のメモリにストアするのは,

MOV　[0201H], AL

となります. オフセットアドレスは2バイトですから,「201_H」は4桁で「0201_H」とした方がわかりやすいので, 頭に0を付けて表します.

さてここで, オフセットアドレスに付いている [] に注目してください. 先の命令では「FF_H」という値そのものが直接ALレジスタにロードされることを表していますが, [0201H] という表現では「0201_H」という値そのものではなく, オフセットアドレス「0201_H」で指し示すメモリの内容という意味を表しています. たとえば, 以下のような例を見てみましょう.

MOV　AL , [0201H] … [] が付くと間接的表現となり, 値「0201_H」そのものではなく, 値「0201_H」によって指定されたオフセットアドレスのメモリの内容をALレジスタにロードする.

MOV　AL , 0FFH … [] がないと直接的な表現となり, 値「FF_H」そのものをALレジスタにロードする.

[] が付くと [] の中身そのものではなく, 中身が指し示すアドレスのメモリの内容が対象, []がないとその数値そのものが対象となるのです. まったく意味が異なるので注意してください.

---

* アセンブリ言語では16進数で先頭に英文字がくる場合, 先頭に0を付けてラベル (8章で解説) と区別する必要がある.

この違いは，5.6 章で解説したプログラム実行のメカニズムから考えるとよくわかります．以下に両者の違いを図解しておきましょう（図 6-3）．

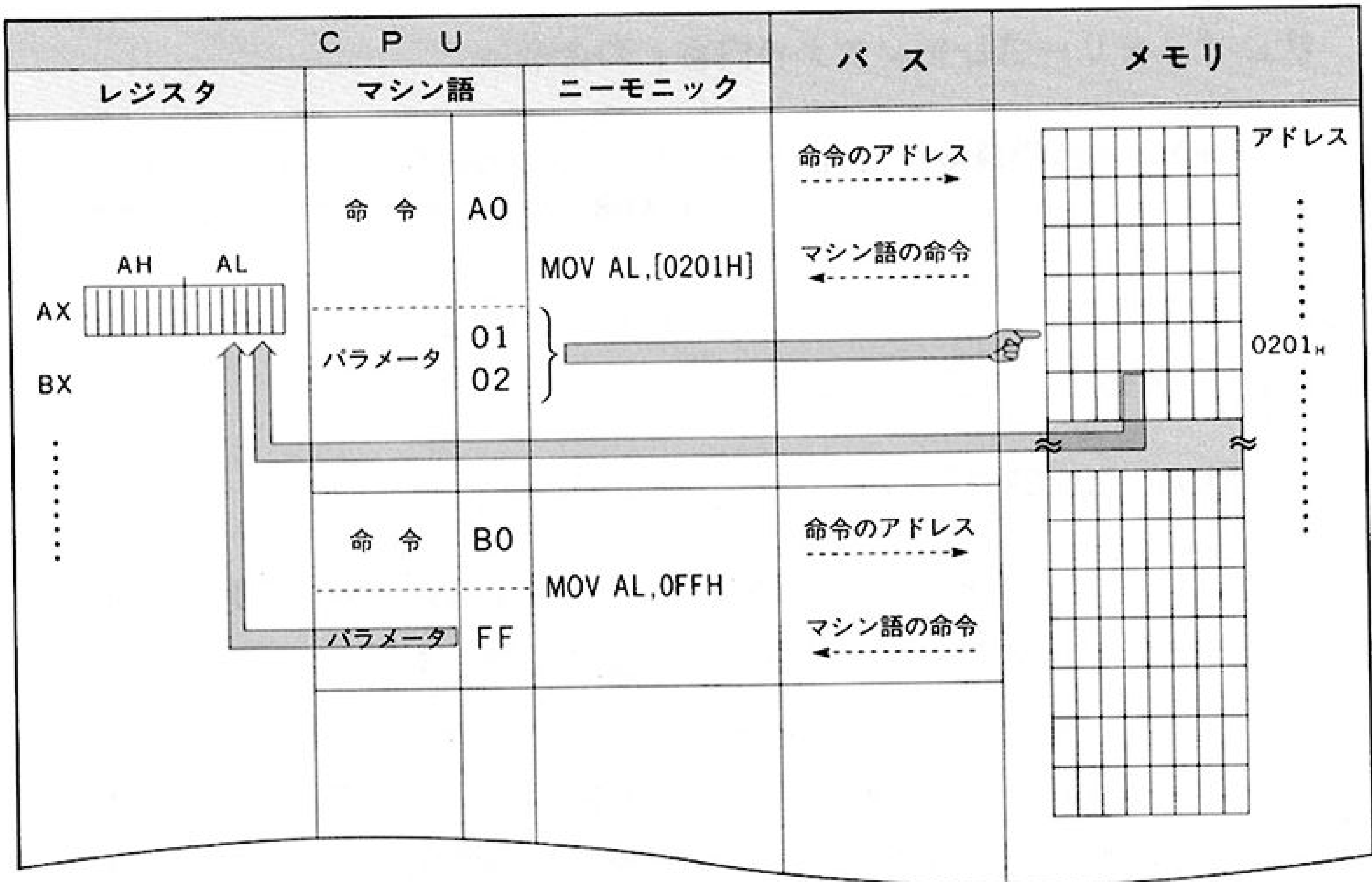


図 6-3 「MOV AL ,[0201H]」と「MOVE AL , 0FFH」の違い

これまでのことをまとめた実習プログラムを次に示します．

MOV AL , 0FFH ……… AL レジスタに 1 バイトデータ「FF_H」をロードする．

MOV [0201H] , AL … AL レジスタの内容をオフセットアドレス「0201_H」のメモリにストアする．

このプログラムを DEBUG 上で入力し，実行して結果を確かめてみましょう．アセンブリ言語では値が 16 進数であることを示すために数値の後に「H」を付けなければなりませんが，DEBUG ではすべての値が 16 進数として扱われるのでわざわざ「H」を付けなくてよい（付けてはいけない*）ことに注意してください．

* SYMDEB では 16 進数を表すのに「H」を付けることもできる．

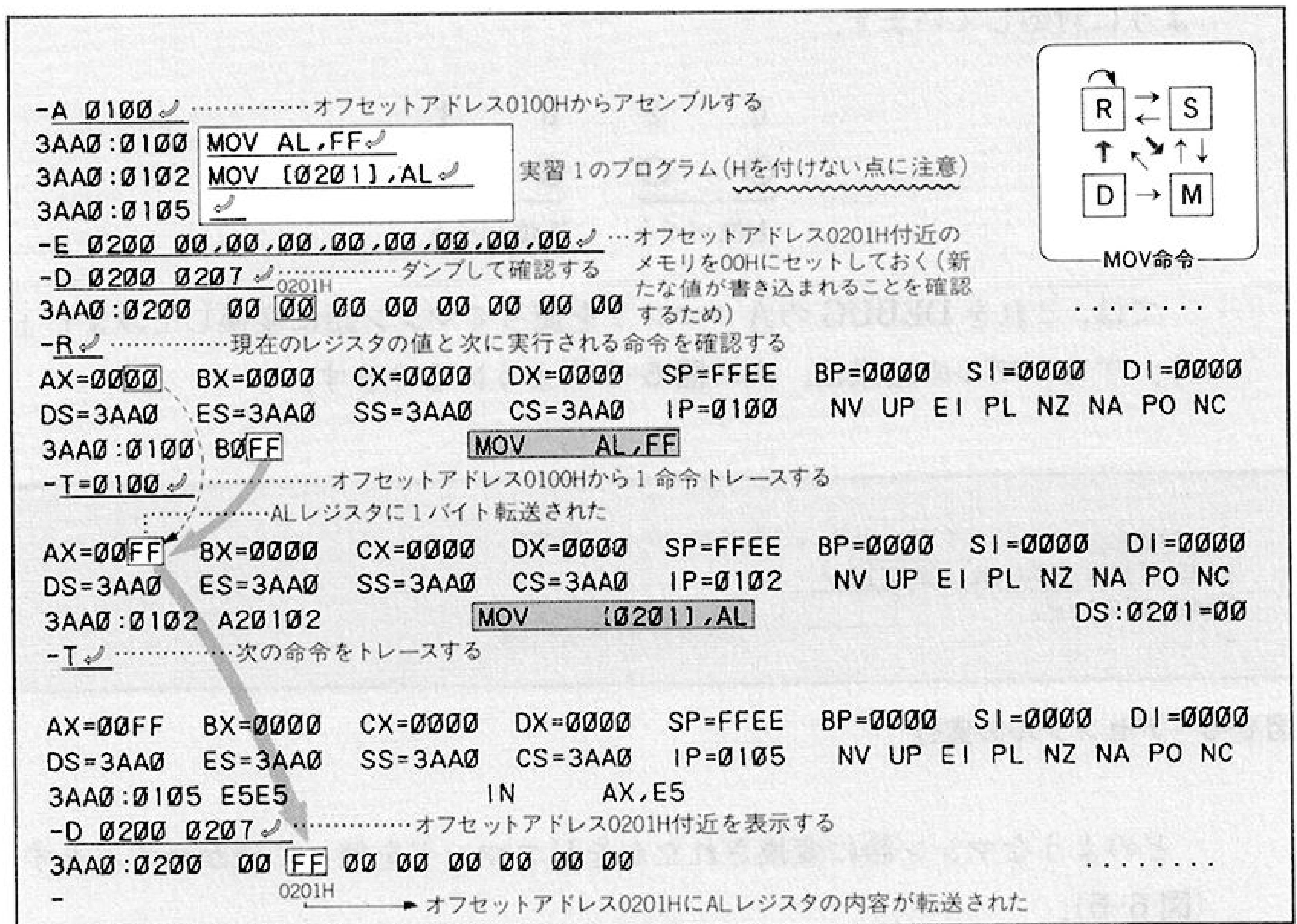


図6-4　実習1の実行結果

実習2　メモリ間の1バイトデータの転送

メモリ上の任意のアドレスの1バイトデータをALレジスタにロードし，そのまま別のアドレスのメモリにストアしてみましょう．

「ALレジスタ←メモリ内容」の命令は，

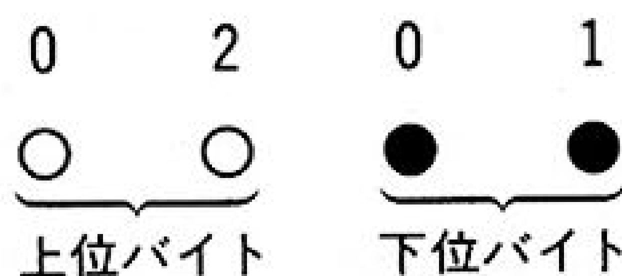
MOV　AL，[○○●●] …… オフセットアドレス「○○●●」で指し示すメモリの内容をALレジスタにロードする．

という形式で表します．たとえばオフセットアドレス「0201_H」で示されるメモリの内容をALレジスタにロードするのは，

MOV　AL，[0201H]

となります．この例では，2バイトの値「0201_H」と「○○●●」とは，次の

ように対応しています．



では，これをDEBUGのAコマンドを使ってマシン語に変換してみましょう．アセンブルの結果は，次の図6-5のようになります．

```
-A Ø1ØØ↵ ……………オフセットアドレス0100Hからアセンブルする
3AAØ:Ø1ØØ MOV AL,[Ø2Ø1]↵
3AAØ:Ø1Ø3 ↵           └──── Hを付けないことに注意
-
```

図6-5　アセンブルの実行

どのようなマシン語に変換されたかをUコマンドを使って確かめてみます（図6-6）．

```
-U Ø1ØØ Ø1Ø2↵ ……………オフセットアドレス0100Hから0102Hまでを逆アセンブルする
3AAØ:Ø1ØØ A0Ø1Ø2 ……        MOV    AL,[Ø2Ø1]
-                  :……このようなマシン語に変換されている
            02 01 ……………1ワードの上位バイトと下位バイトが逆転している
              ⇩
           パラメータ
```

図6-6　逆アセンブルの実行

図6-6のように，オフセットアドレスの2バイトの上位と下位が逆になっていることに注意してください．このことはすでに何度か見ているので疑問に思っていた方もいるかもしれません．マシン語としてメモリに置かれている状態では，アドレスやデータなどの2バイトの値が「○○●●」であればメモリ上では「●●○○」のように，必ず上位と下位のバイトが入れ替わります．これは80系CPUの特徴となっています．

次は，ALレジスタのデータを任意のアドレスのメモリにストアする命令を実習してみましょう．

実習1でも解説したように，「メモリ←ALレジスタの内容」の命令は，

MOV　[○○●●]，AL

という形式で表します．ここでも「○○●●」はマシン語としてのメモリ上の配置は，「●●　○○」と上位／下位のバイトが逆になります．このことは，転送命令に限りません．すべての2バイトのデータはメモリ上では逆になると思ってください．

以上より，「実習2」は次のようなプログラムになります．

MOV　AL，[0201H]……オフセットアドレス「0201H」のメモリの内容をALレジスタにロードする．

MOV　[0401H]，AL……ALレジスタの内容をオフセットアドレス「0401H」のメモリにストアする．

上の2ステップのプログラムをDEBUG上でアセンブルし，実行して結果を確かめてみましょう．実行結果は，次の図6-7のようになります．

```
-A 0100       ……オフセットアドレス0100Hからアセンブルする
3AA0:0100 MOV AL,[0201]
3AA0:0103 MOV [0401],AL      実習2のプログラム
3AA0:0106
-E 0200 12,34      ……オフセットアドレス0200H付近に適当な値をセットしておく
-D 0200 0201       ……その値を確認する
                0201H
3AA0:0200   12 34
-E 0400 00,00,00,00,00,00,00,00   ……オフセットアドレス0401H付近に00Hをセット
                                      しておく（新たな値が書き込まれることを確
                                      認するため）
-D 0400 0407       ……その値を確認する
3AA0:0400   00 00 00 00 00 00 00 00
                0401H
-R ……現在のレジスタの値と次に実行される命令を確認する
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3AA0  ES=3AA0  SS=3AA0  CS=3AA0  IP=0100   NV UP EI PL NZ NA PO NC
3AA0:0100 A00102        MOV     AL,[0201]                     DS:0201=34
-T=0100  ……オフセットアドレス0100Hから1命令トレースする
          オフセットアドレス0201Hのメモリの内容がALレジスタに転送された
AX=0034  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3AA0  ES=3AA0  SS=3AA0  CS=3AA0  IP=0103   NV UP EI PL NZ NA PO NC
3AA0:0103 A20104        MOV     [0401],AL                     DS:0401=00
-T  ……次の命令をトレースする
```

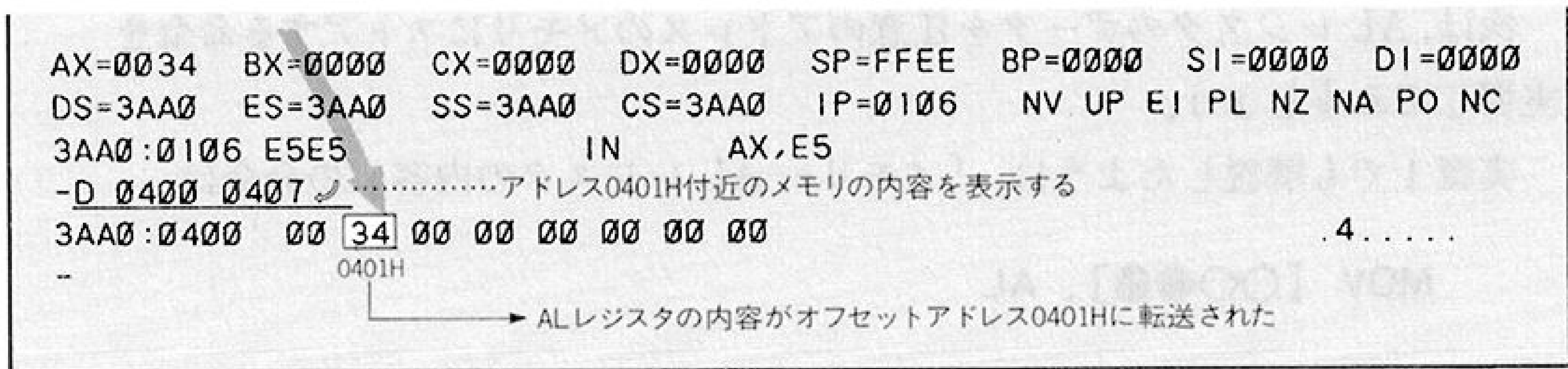


図 6-7　実習 2 の実行結果

実習 3　1 ワードデータのロード／ストア

任意の 1 ワードデータ (0000_H～$FFFF_H$) を AX レジスタにロードし，そのまま任意のアドレスのメモリにストアしてみましょう．

「AX レジスタ←1 ワードデータ」の命令は，

MOV AX , ○○●● …… 1 ワードデータ「○○●●」を AX レジスタにロードする．

という形式で表します．たとえば 1 ワードのデータ「$FD26_H$」を AX レジスタにロードするのは，

MOV AX , 0FD26H

となります．実習 1 と比べてみると，レジスタの名前が AL から AX に変わり，ロードできる値の範囲が 1 ワード (2 バイト) で表現できる「0000_H～$FFFF_H$」になりました．

転送するデータが 1 バイトであるか 1 ワードであるかは，使用するレジスタが何であるかで決まります．8 ビットのレジスタを指定すれば 1 バイトのデータ転送になり，16 ビット (2 バイト) のレジスタを指定すれば 1 ワードのデータ転送になります．レジスタについては，97 ページの図 5-6「8086CPU のレジスタ」を参照してください．

上の例のように，AX レジスタに「$FD26_H$」をロードすると，AH レジスタに上位バイトの「FD_H」がロードされ，AL レジスタに下位バイトの「26_H」がロードされます (図 6-8)．

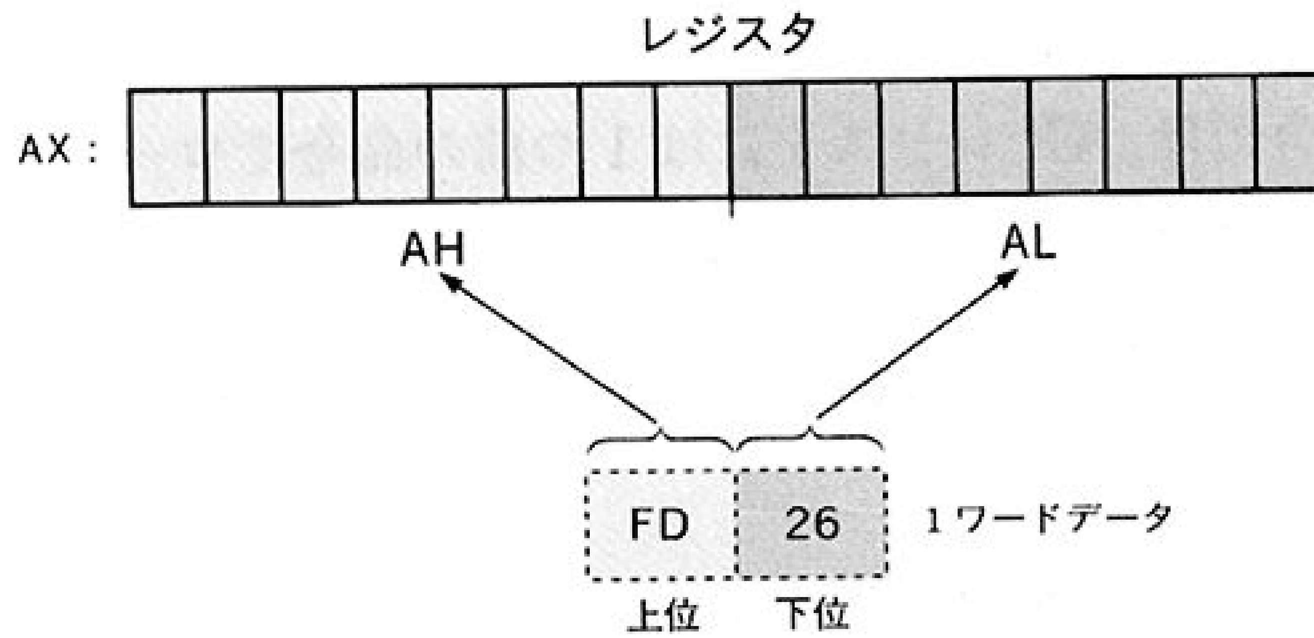


図 6-8　AX レジスタに「FD26H」をロードする

したがって，

MOV　AX ，○○●● ＜ MOV　AH ，○○ / MOV　AL ，●●

となることがわかるでしょう．

次に，「メモリ← AX レジスタの内容」の命令は，

MOV ［○○●●］，AX …… AX レジスタの内容をオフセットアドレス「○○●●」のメモリにストアする．

という形式で表します．たとえば AX レジスタの内容をオフセットアドレス「0420_H」のメモリにストアするのは，

MOV ［0420H］，AX

となります．

実習 2 で説明したように，1 ワードのデータがメモリ上に置かれる場合には，その上位と下位が逆になります．この命令では，下位バイトが指定されたオフセットアドレスのメモリに入り，上位バイトがその次のメモリに入ります．つまり，

MOV	［□□■■］		，	AX
アドレス	□□■■	□□■■＋1	←	AX
	●●（下位）	○○（上位）		○○●●（上位 下位）

という形で上位と下位が逆転します.

上の例の場合では, AX レジスタには1つ前の命令でロードした値「FD26H」が入っているので,

MOV　[0420H]　,　AX

アドレス	0420	0421		AX
メモリ内容	26 (下位)	FD (上位)	←	FD (上位) 26 (下位)

となります. これを図にして示してみましょう (図6-9).

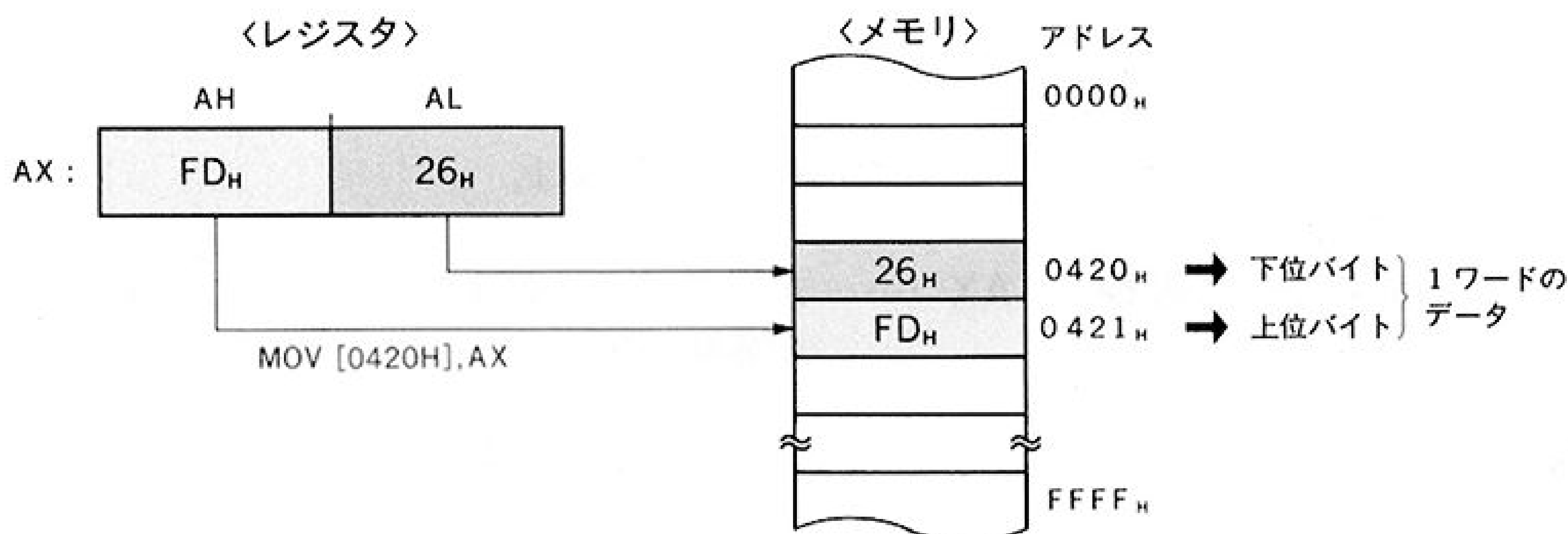


図6-9　オフセットアドレス「0420_H」から始まるメモリにAX レジスタの内容をストアする

メモリから1ワードデータをレジスタにロードする場合も同様に上位と下位が逆転します. つまり, ストアするときに上位と下位が逆になり, ロードするときにまた上位と下位が逆転するので元に戻ることになります(図6-10).

以上により, 「実習3」のプログラムをまとめて示してみましょう.

MOV　AX　,　FD26H　……　AX レジスタに1ワードデータ「$FD26_H$」をロードする.

MOV　[0420H] ,　AX　…　AX レジスタの内容をオフセットアドレス「0420_H」のメモリにストアする.

このプログラムを DEBUG 上で入力し, 実行して結果を確かめてみます(図6-11).

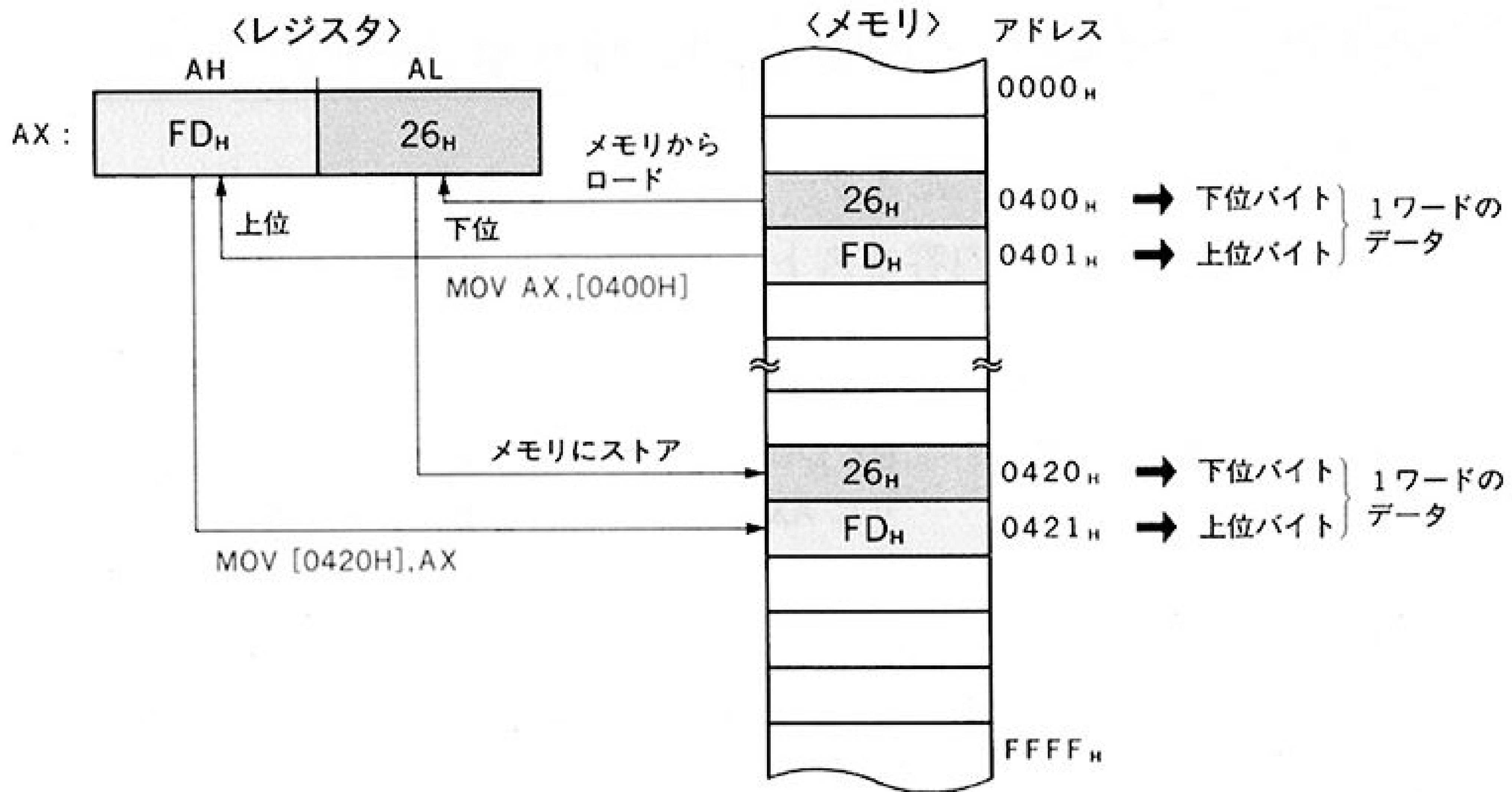


図 6-10　1ワードデータの「ロード」と「ストア」

```
-A 0100 ……………オフセットアドレス0100Hからアセンブルする
3AA0:0100 MOV AX,FD26
3AA0:0103 MOV [0420],AX        実習3のプログラム
3AA0:0106
-E 0420 00,00,00,00,00,00,00,00 …オフセットアドレス0420H付近の
                                  メモリに00Hをセットしておく
-D 0420 0427 ……………その値を確認する
3AA0:0420 00 00 00 00 00 00 00 00                               ........
          0420H 0421H
-R
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3AA0  ES=3AA0  SS=3AA0  CS=3AA0  IP=0100   NV UP EI PL NZ NA PO NC
3AA0:0100 B826FD           MOV     AX,FD26
-T=0100 ……………オフセットアドレス0100Hから1命令トレースする
             AXレジスタに1ワードデータFD26Hが転送された
AX=FD26  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3AA0  ES=3AA0  SS=3AA0  CS=3AA0  IP=0103   NV UP EI PL NZ NA PO NC
3AA0:0103 A32004           MOV     [0420],AX                DS:0420=0000
-T ……………次の命令をトレースする

AX=FD26  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3AA0  ES=3AA0  SS=3AA0  CS=3AA0  IP=0106   NV UP EI PL NZ NA PO NC
3AA0:0106 E5E5             IN      AX,E5
-D 0420 0427 ……………オフセットアドレス0420H付近のメモリの内容を確認する
3AA0:0420 26 FD 00 00 00 00 00 00                               &}......
          0420H 0421H
-
          └──→ 上位バイトと下位バイトが逆転してメモリに転送される
```

R ⇄ S, D → M　MOV命令

図 6-11　実習3の実行結果

実習４　BXレジスタでオフセットアドレスを指定されたメモリへのストア

BXレジスタにセットされた１ワードのデータをオフセットアドレスとするメモリに，AXレジスタの内容をストアしてみましょう．

この命令は，

MOV　[BX]，AX ……　BXレジスタの内容をオフセットアドレスとするメモリにAXレジスタの内容をストアする．

という形式で表します．[] を使うと [] 内のレジスタの値ではなく，その値をオフセットアドレスとする「メモリの内容」を指し示すことは「実習１」で解説したとおりです．

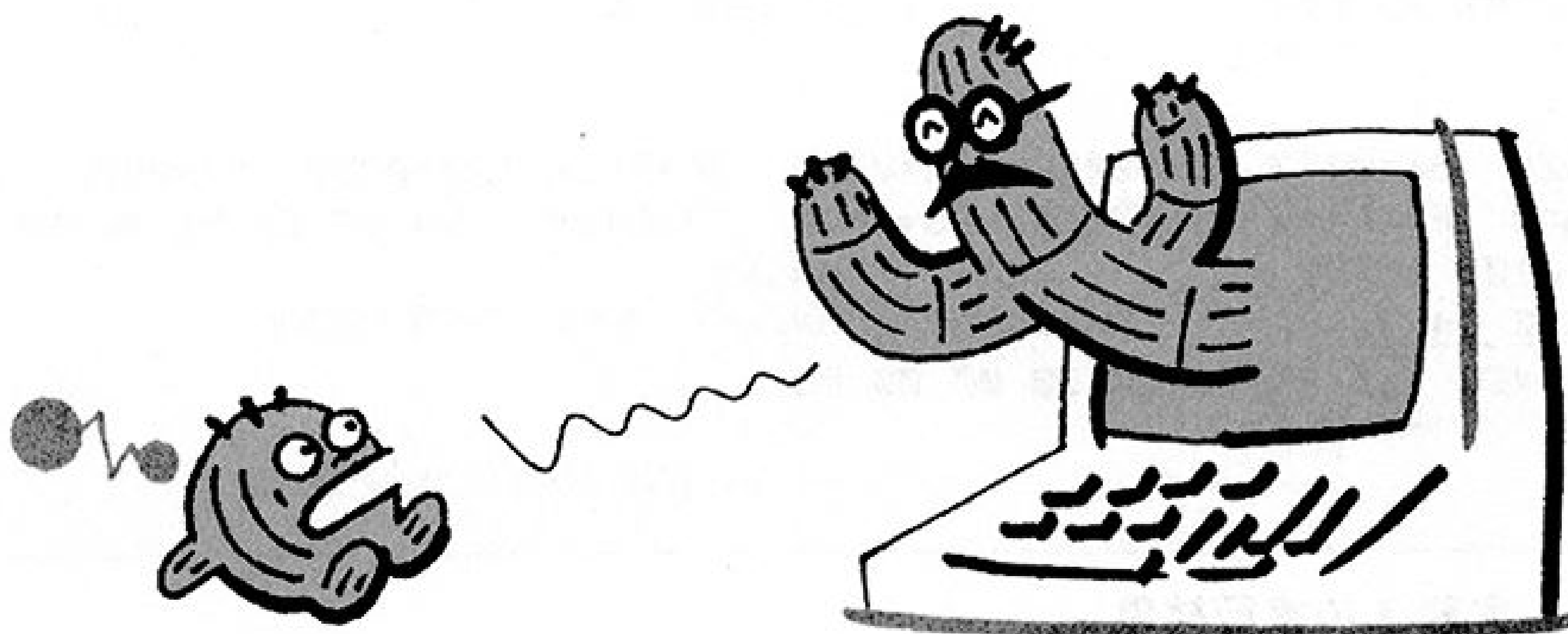
具体的に，次のようなプログラムを作ってみましょう．BXレジスタに１ワードデータ「0430_H」をロードしておき，AXレジスタにはメモリにストアするデータとして「$7FE0_H$」をロードしたあと，前述のメモリへの転送命令を実行します．

このプログラムを次に示します．

MOV　BX，0430H ……　BXレジスタに１ワードデータ「0430_H」をロードする（オフセットアドレス）．

MOV　AX，7FE0H ……　AXレジスタに１ワードデータ「$7FE0_H$」をロードする（メモリにストアするデータ）．

MOV　[BX]，AX ………　BXレジスタでオフセットアドレスを指定されるメモリにAXレジスタの内容をストアする．



これをDEBUG上で入力し，実行して結果を確認してみましょう(図6-12).

図6-12　実習4の実行結果

実習1～4はデータ転送命令のほんの一例です．本章の実習が一通り終わったら，レジスタ名を変えたりデータの数値を変えたりして，各自でいろいろと試してみてください．

アドレッシングモード
—データ転送が可能なレジスタとメモリの組合せ

データの格納場所には，レジスタとメモリの2種類があります．レジスタの中でもセグメントレジスタは「5.5　セグメントの考え方」で解説したように特別な役割があり操作が制限されているので，他のレジスタと区別して扱います．データ転送はこの三者，つまりレジスタ*，セグメントレジスタ，メモリという3種類のデータ格納場所の間で行われると考えられます．さらに，データ転送元，つまり右側に指定されるのは，データが格納されている場所とは限らず，データそのものという場合もあります．

データ転送は，3種類のデータ格納場所(レジスタ，セグメントレジスタ，メモリ）およびデータそのものをさまざまに組み合わせて行うことができます．たとえば「実習1」のプログラムは，

MOV　AL，0FFH　……ALレジスタに「FF_H」をロードする．

MOV　[0201H]，AL　……オフセットアドレス「0201_H」のメモリにALレジスタの内容をストアする．

というものでしたが，実はこれは次のように1命令で書くこともできます．

MOV　[0201H]，0FFH**……オフセットアドレス「0201_H」のメモリに1バイトのデータ「FF_H」をストアする．

つまりALレジスタを経由せずに，直接メモリにデータを書き込むことが可能です．これに対し「実習2」は，

MOV　AL，[0201H]　……ALレジスタにオフセットアドレス「0201_H」のメモリの内容をロードする．

MOV　[0401H]，AL　……ALレジスタの内容をオフセットアドレス「0401_H」のメモリにストアする．

というプログラムでしたが，これを，

---

*フラグレジスタ，IP(インストラクションポインタ)レジスタは，データ転送命令で直接操作できないので，ここでいうレジスタの中には含まない．

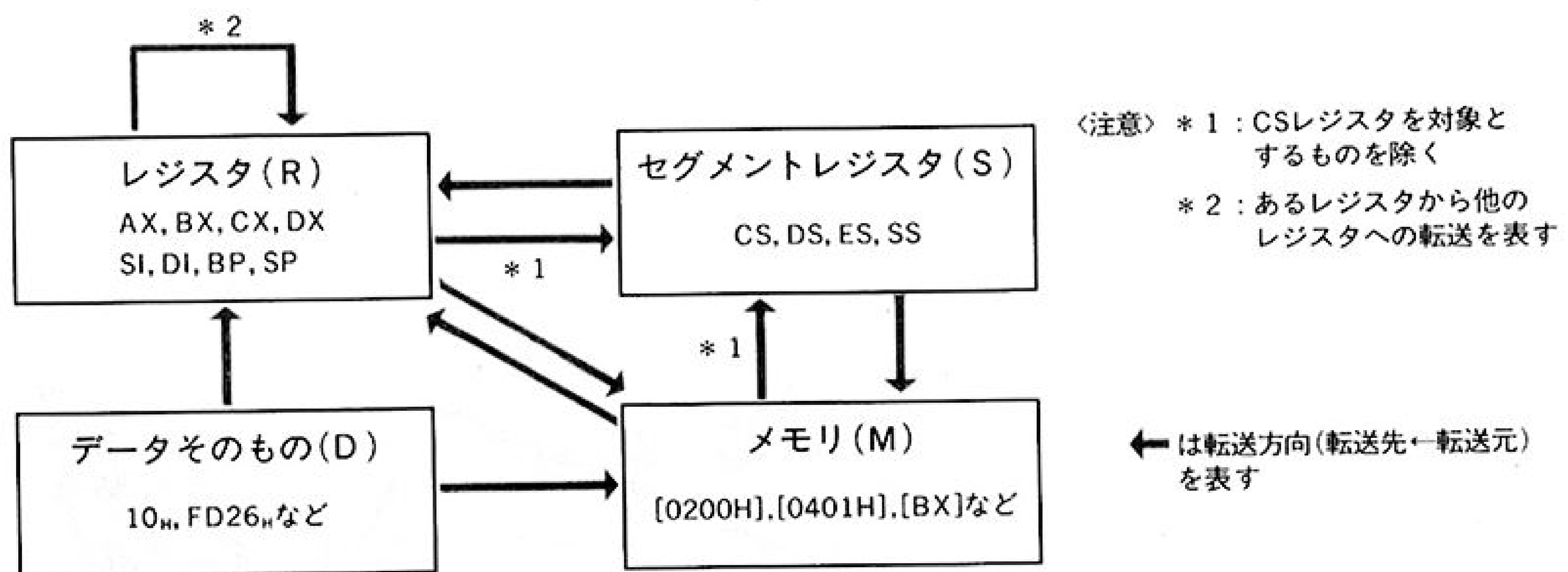
**正確には，「MOV BYTE PTR [0201H]，0FFH」でなければならない．くわしくは155ページからの解説を参照．

MOV　[0401H]，[0201H] ……　**オフセットアドレス「0201_H」のメモリの内容をオフセットアドレス「0401_H」のメモリにストアする．**

と書くことはできません．メモリに直接値をストアすることはできても，メモリからメモリへ直接データを転送することはできないのです．このような場合は，やはり「実習2」のようにレジスタを経由して転送する必要があります．

このように，レジスタ，セグメントレジスタ，メモリ，データそのものという4つの組合せのうち，データを転送できる組合せは決まっています．この組合せのことを**アドレッシングモード**と呼びます．アドレッシングモードとして許されている組合せ以外では，データを転送することはできません．それ以外の場合は前述のようにレジスタを経由するなどして転送しなければなりません．

データ転送命令についてアドレッシングモードとして用意されている組合せを次の図6-13にまとめておきます．この図で矢印が向かっている方向についてのみデータ転送を行うことができます．



以降の実習では，マシン語命令のアドレッシングモードを右図のように略号で表すことにする．ここで細線は，そのマシン語命令で使用できるアドレッシングモード，太線がその実習で使用したアドレッシングモードを意味する．

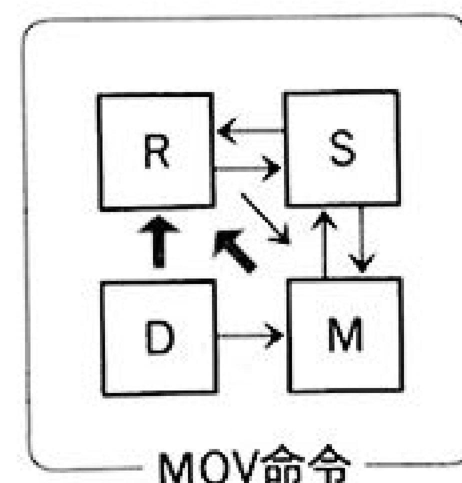
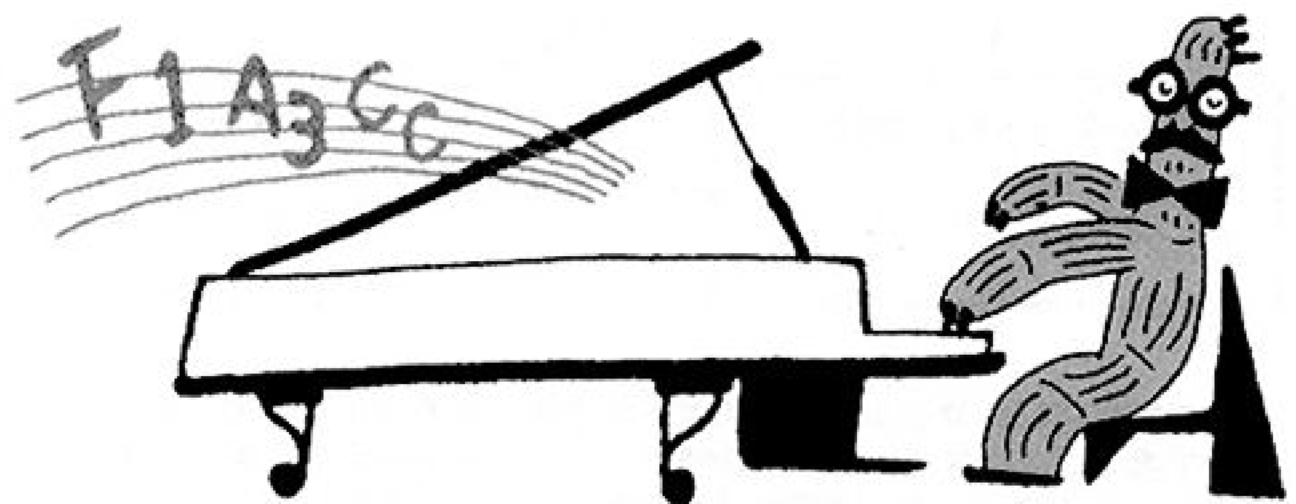


図6-13　データ転送命令のアドレッシングモード

この図でレジスタについては自分自身に矢印が戻っていますが，これはあるレジスタから別のレジスタにデータ転送ができることを意味しています．それに対しメモリからメモリへは矢印がありません．つまり，メモリからメモリへのデータ転送はできないのです*．

データそのものからセグメントレジスタにも矢印がありません．セグメントレジスタは不用意に値をセットされては困るので，直接値を入れることができないようになっています．

データ転送命令以外の命令についても，操作の対象となるレジスタやメモリの組合せはアドレッシングモードとして決められています．以下の節では，各命令の実習ごとにアドレッシングモードを略図で示しておきますので，参照してください．具体的な組合せの例は APPENDIX の命令一覧表に示していますが，この図を見るだけでもどういう組合せが可能かがわかるでしょう．



---

* MOVS 命令は唯一メモリ間の転送命令であるが，ここでは例外として取り上げない．

レジスタを使ったアドレス指定　―ポインタ―

「実習4」ではBXレジスタの内容をオフセットアドレスとするメモリへAXレジスタの内容をストアするというプログラムを作りました．「5.3 レジスタとその機能」でも簡単に解説しましたが，BXレジスタをこのような目的に使用することを，ポインタとして使用すると言います．これは特定のメモリを指し示し，そのメモリの内容を処理の対象にするという意味です．

BX以外のレジスタにもポインタとして使用できるレジスタがありますが，どのレジスタでも可能なわけではありません．ポインタとして使用できるのは「BX，BP，SI，DI」の4つのレジスタです．

たとえば，

```
MOV  AX ,[SI] …… AXレジスタにSIレジスタの指し示す1ワードのメモリの内容をロードする．
```

のように使うことができます．もちろん，このときポインタとするレジスタには，目的とするメモリのオフセットアドレスが入っていなければなりません．ポインタを使うと，たとえばポインタのレジスタの値を順々に変化させることによって連続したメモリを次々とアクセスするプログラムを書くことができます（このようなポインタを使った実習は「実習11」と「実習13」で行います）．

これらのレジスタをポインタとして使用する場合には，レジスタの内容にさらにある値を加えた値をオフセットアドレスとすることもできます．たとえば，

```
MOV  AX ,[BX+5]
```

のように指定します．もうわかると思いますが，[BX+5]は「BX(の内容)＋5」という値をオフセットアドレスとするメモリの内容を表しています．さらにレジスタとレジスタの値を加えてアドレスを指定することもできます．ポインタとして使用可能なレジスタのすべての組合せを次ページの**表6-2**に挙げておきましょう．

なお，ここで解説したレジスタをポインタとしてメモリを指定する方法はデータ転送命令に限らず，すべてのメモリを対象とする命令（算術演算命令，論理演算命令，ローテート・シフト命令など）で使うことができます．

レジスタとディスプレースメントの組合せ	ニーモニック	指定例
ベースレジスタ（BX / BP）→ ディスプレースメント（指定しない / d）	[BX]	—
	[BX+d]	[BX+6]
	[BP]	—
	[BP+d]	[BP−5]
インデックスレジスタ（SI / DI）→ ディスプレースメント（指定しない / d）	[SI]	—
	[SI+d]	[SI+100H]
	[DI]	—
	[DI+d]	[DI+7]
ベースレジスタ（BX / BP）→ インデックスレジスタ（SI / DI）→ ディスプレースメント（指定しない / d）	[BX+SI]	—
	[BX+SI+d]	[BX+SI+0FD00H]
	[BX+DI]	—
	[BX+DI+d]	[BX+DI+0200H]
	[BP+SI]	—
	[BP+SI+d]	[BP+SI+0DH]
	[BP+DI]	—
	[BP+DI+d]	[BP+DI−7]

〈注意〉
・ディスプレースメントは，レジスタで指定された内容からどれだけ離れているか(変位)を表す値である．
・dは1バイトまたは1ワードのデータを表す．

表 6-2　ポインタとして使用できるレジスタの組合せ

6.3 算術演算命令

CPUにはレジスタやメモリに対して，足し算や引き算，掛け算，割り算といった四則演算，それに論理演算など各種の演算を行う機能が用意されています．

各種演算は対象となるレジスタやメモリの中で行われるのではなく，実際にはCPU内部のALU（Arithmetic and Logic Unit）と呼ばれる演算部で行われます．演算の対象となるレジスタやメモリからデータがALUに転送され，演算が実行された後に，その結果がレジスタやメモリに転送されるのです．さらにその時，演算の結果そのものだけでなく，結果が0であったか，負であったかといった状態がフラグレジスタにセットされます．

私たちが実際にプログラムを作る場合には，CPU内部の動作は直接には関係ないので，演算はALU上ではなくレジスタやメモリ上で行われると考えてもらってさしつかえありません．ただし，結果そのものだけでなく，その状態もフラグレジスタに保存されるということは，重要なことなのでぜひ理解しておいてください．ALUの働きを図解したものを図6-14に示しておきます．

この図に示すようなXとYの組合せとして可能なものは，前節のデータ転送命令と同じくアドレッシングモードとして決められています．以下の実習では，前節の図6-13と同じ略号を用いた図で，各命令のアドレッシングモードを挙げておきます．矢印はYからX，つまり演算が行われて値が変化するレジスタやメモリの方向を指しています．なお，演算の対象が1つしかない命令もあります．その場合には矢印の出発点はありません．

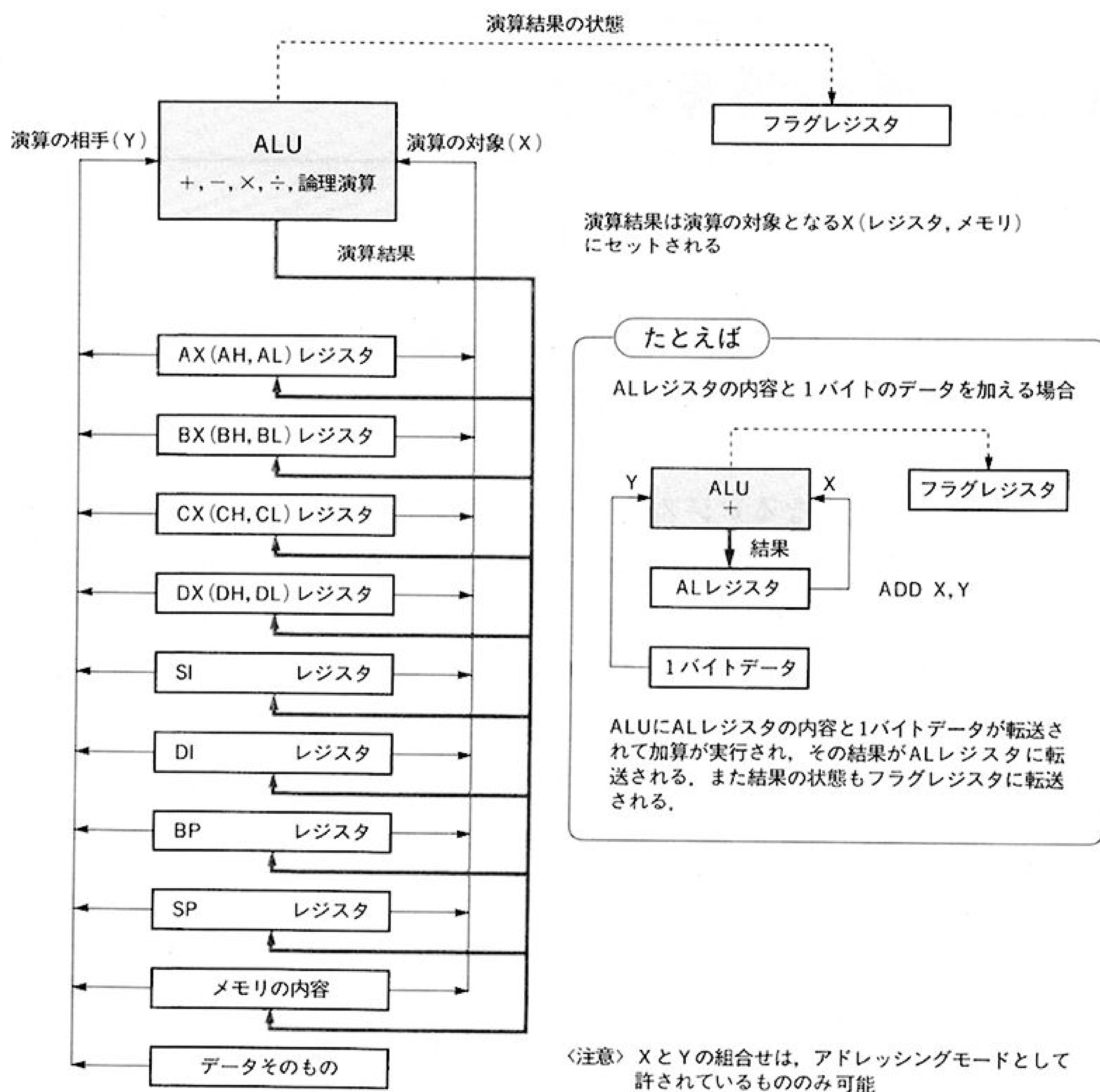


図 6-14　ALU の働き

加算／減算命令

それでは，まず始めに加算／減算命令から実習していくことにしましょう．

実習 5　1バイトデータの加減算

AL レジスタと1バイトデータとの加算および減算の実習を行います．AL レジスタの内容に任意の1バイトデータを加え，その結果を適当なオフセッ

トアドレスのメモリにストアします．続けてその値から任意の1バイトデータを引き，その結果を次のアドレスにストアしてみましょう．

ALレジスタと任意の1バイトデータの加減算命令は，

ADD AL，●●（アド(ADDition)）……ALレジスタの内容に1バイトデータ「●●」を加算する．
SUB AL，●●（サブトラクト(SUBtruct)）……ALレジスタの内容から1バイトデータ「●●」を減算する．

という形式で表します．これらの演算結果は，演算の対象であるALレジスタにセットされます．

実習5のプログラムを次に示します．最初のALレジスタの内容を「10_H」，加算する任意の1バイトのデータを「40_H」，減算する1バイトデータを「20_H」，ストアするメモリのオフセットアドレスを「0240_H」としておきましょう．

MOV AL，10H ………ALレジスタに1バイトデータ「10_H」をロードする．
ADD AL，40H ………ALレジスタの内容に1バイトデータ「40_H」を加える．
MOV [0240H]，AL …その結果がセットされているALレジスタの内容をオフセットアドレス「0240_H」のメモリにストアする．
SUB AL，20H ………ALレジスタの内容から1バイトデータ「20_H」を引く．
MOV [0241H]，AL …その結果を次のアドレスにストアする．

このプログラムをDEBUG上で入力し，実行して結果を確かめてみます（図6-15）．実習1～4ではT (Trace) コマンドを使って1命令ずつ実行してきましたが，これ以降の実習ではBASICの「RUN」に相当するG (Go) コマンドを使って実習を行うことにします．Gコマンドの書式は以下のとおりです．

G=<アドレス1> <アドレス2> ……アドレス1からアドレス2までのプログラムを実行する．

Gコマンドはアドレス1からアドレス2までのプログラムを一気に実行します．ただし停止するのは，アドレス2の命令の直前の命令を実行した後であることに注意してください．アドレス2の命令は実行されません．

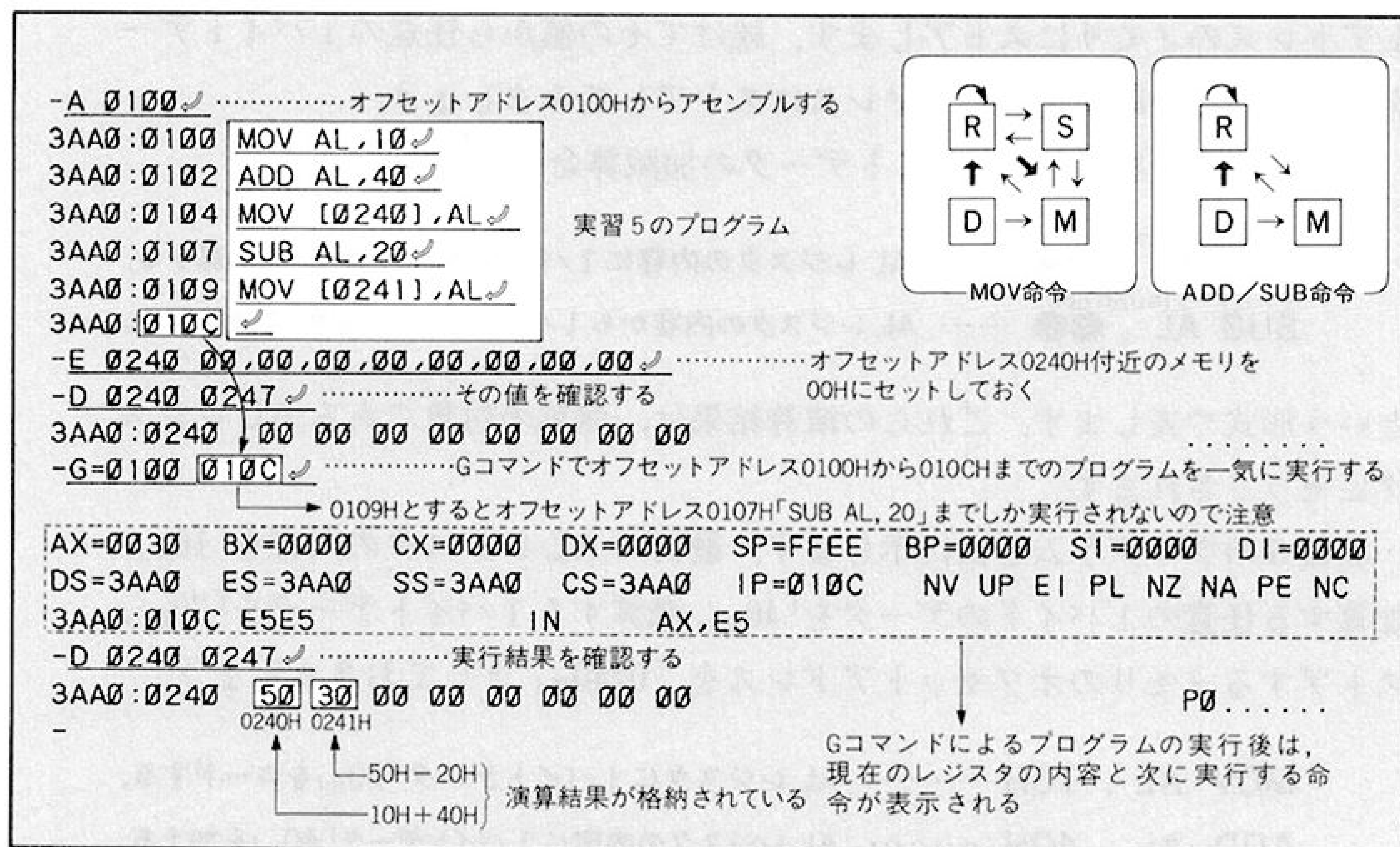


図 6-15　実習 5 の実行結果

実習6　各レジスタのインクリメント，デクリメント

各レジスタの値を＋1，または－1する命令の実習です．これは，実習5で行ったADD命令／SUB命令を使えばもちろんできますが，＋1と－1に関しては，それ専用の命令が用意されています．

ここで，＋1することを**インクリメント**，－1することを**デクリメント**と呼びます．これはCPUの命令では「INC」と「DEC」となります．たとえば，

INC　AL（＝ADD　AL，01）

であり，どちらもALレジスタの内容に1を加算します．両者が異なる点は，INC命令の方がマシン語命令が短くてすみ，実行が速いということです*．またフラグの変化にも若干違いがあります．マシン語のプログラミングでは

*アドレッシングモードによって異なるが，INC／DEC命令を使うとADD／SUB命令で1を足したり引いたりするよりも1バイトから3バイト，マシン語命令のバイト数が短かくなる．なお，マシン語命令のバイト数が短ければ必ず実行速度が速い（クロック数が少ない）というわけではないので注意すること．

+1したり−1したりする必要がひんぱんに起こるので，ほとんどのCPUでこの命令が用意されています．

今回の実習では，BLレジスタに任意の1バイトデータをロードし，それをALレジスタに転送したあと，ALレジスタを−1します．そして，そのALレジスタの内容をメモリにストアします．続けてBLレジスタの内容を+1し，そのBLレジスタの内容をメモリにストアします．つまり最初の1バイトデータを−1および+1した値がメモリにストアされるわけです．

新しく登場する命令は次の2種類です．ALレジスタの内容を+1（インクリメント）する命令，−1（デクリメント）する命令はそれぞれ，

INC AL（インクリメント (INCrement)） …… ALレジスタをインクリメントする．
DEC AL（デクリメント (DECrement)） …… ALレジスタをデクリメントする．

という形式で表します．

図6-16のアドレッシングモードの図を見てください．矢印の根元が示されていません．これは演算の対象が1つしかないことを意味しています．

「実習6」のプログラムを次に示しましょう．1バイトのデータを「10_H」，ストアするメモリアドレスは最初のものを「0170_H」，次のものを「0171_H」としておきます．

MOV BL , 10H ……… BLレジスタに1バイトデータ「10_H」をロードする．
MOV AL , BL ………… それをALレジスタに転送．
DEC AL ………………… ALレジスタの内容（現在は「10_H」）をデクリメント（-1）．
MOV [0170H] , AL … その結果をオフセットアドレス「0170_H」のメモリにストア．
INC BL ………………… BLレジスタの内容（現在は「10_H」）をインクリメント（+1）．
MOV [0171H] , BL …… その結果をオフセットアドレス「0171_H」のメモリにストア．

このプログラムをDEBUG上で入力し，実行して結果を確かめてみます．

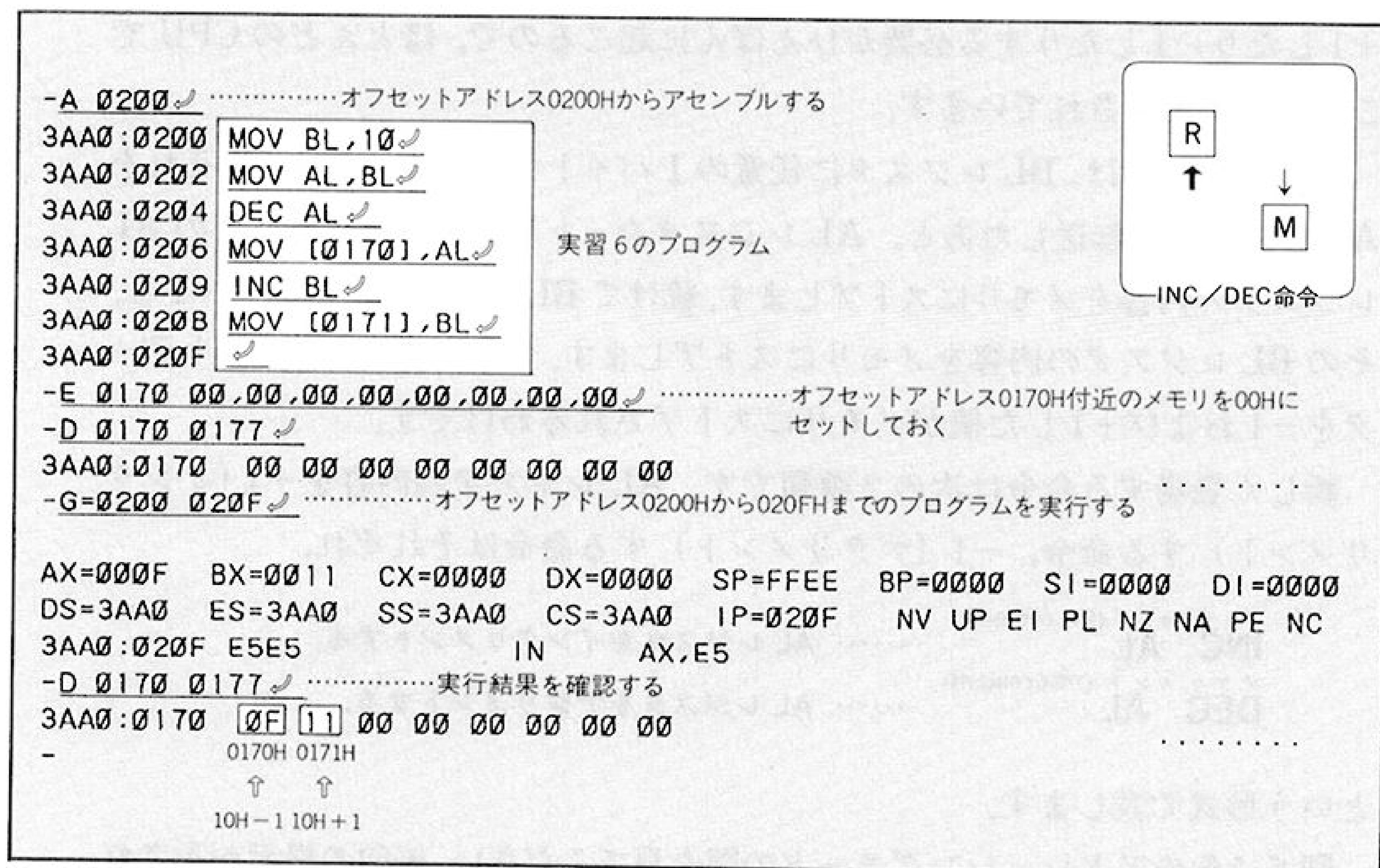


図 6-16　実習 6 の実行結果

比較命令

レジスタやメモリの内容を他のレジスタやメモリ，あるいはデータそのものと比較する命令がコンペア命令です. 具体的には引き算，つまり SUB 命令と同じことをします. ただし，結果は演算の対象となるレジスタやメモリに転送されません. 引き算は行われますが，その結果はどこにも残らないのです. それでは何をするのでしょうか？

図 6-14 で解説したように，演算が行われるときは CPU 内の ALU にデータが転送され，そこで演算が行われます. コンペア命令でもやはり ALU で演算が行われますが，演算結果そのものはもとの場所に転送されずにそのまま捨てられ，その演算結果が 0 であったか負であったかなどという「演算結果の状態」だけがフラグレジスタにセットされるのです. つまり，コンペア命令は各種フラグをセットするだけの命令であるわけです.

この様子を次の図 6-17 に示しますので，通常の演算命令の実行される様子を示した図 6-14 と較べてみてください.

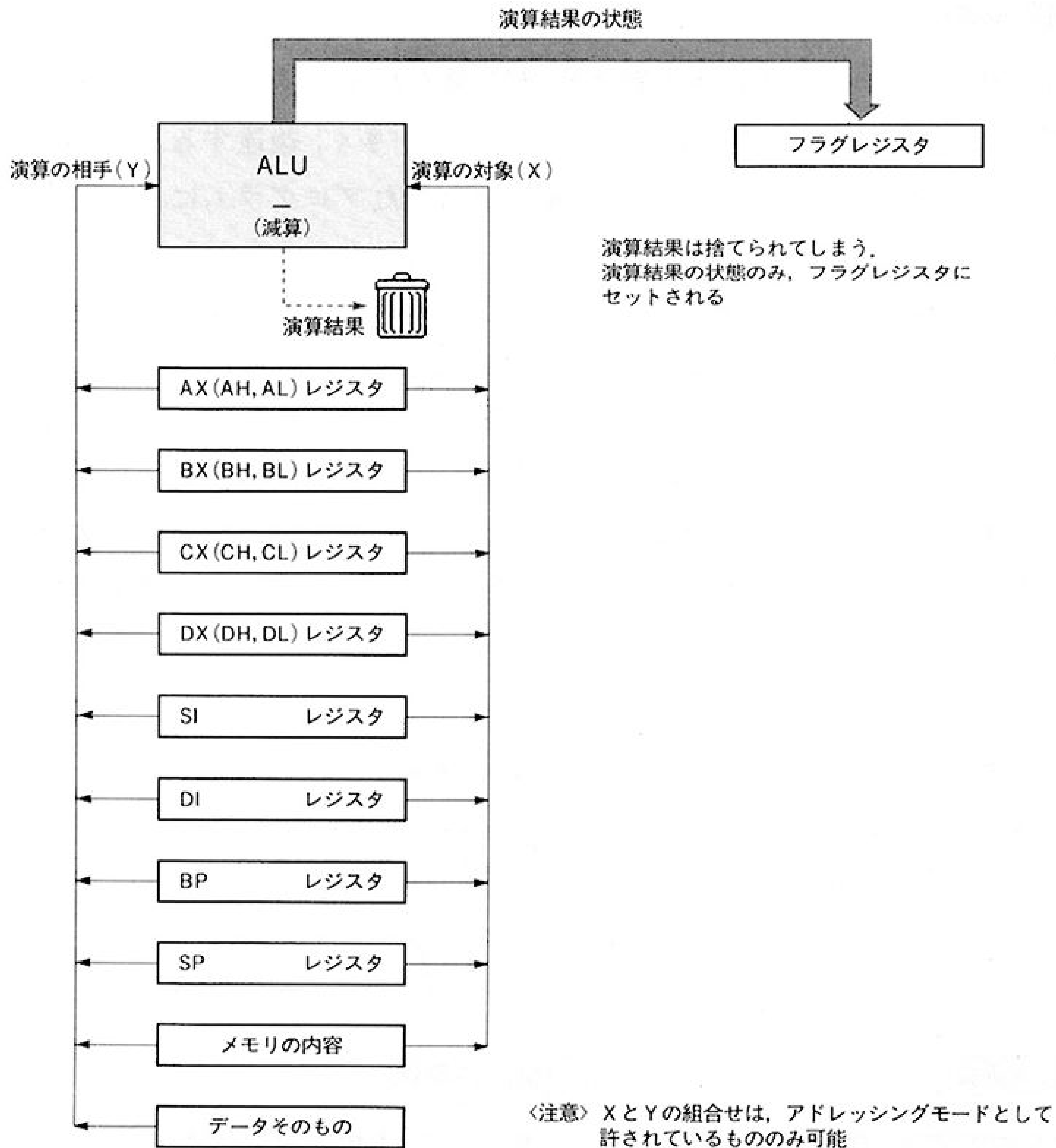


図 6-17　コンペア命令の動作

コンペア命令は，たとえばAXレジスタと1ワードデータとの比較の場合，

コンペア(CoMPare)
CMP　AX　, ○○●● …… **AXレジスタの内容と1ワードデータ「○○●●」を比較し，結果をフラグレジスタにセットする．**

という形式で表します．

コンペア命令は単独では意味をなさず，条件分岐命令と組み合わせて使用するものなので，これを使った実習は，「6.4 フラグと条件分岐命令」の実習9，実習10で取り上げます．

乗除算命令

8086CPU ではマシン語命令で掛け算,割り算を行うことができます.8 ビット CPU では,乗除算の命令を持っていないものが多く,後述するシフト命令（2 倍，1/2 倍を行う命令）と加減算を組み合わせたプログラムによって実現していました.

乗除算命令は他の演算と異なり，使用できるレジスタが決まっており，それ以外のレジスタを使用することはできません.使用できるレジスタは「AX レジスタ」またはその下位バイトである「AL レジスタ」ですが，さらに「DX レジスタ」も使われる場合があります.このレジスタの内容に他のレジスタの内容を掛ける，という形で乗算を，このレジスタの内容を他のレジスタの内容で割る，という形で除算を行います.これについては,以下の図 6-18 を見てもらうとよくわかるでしょう.

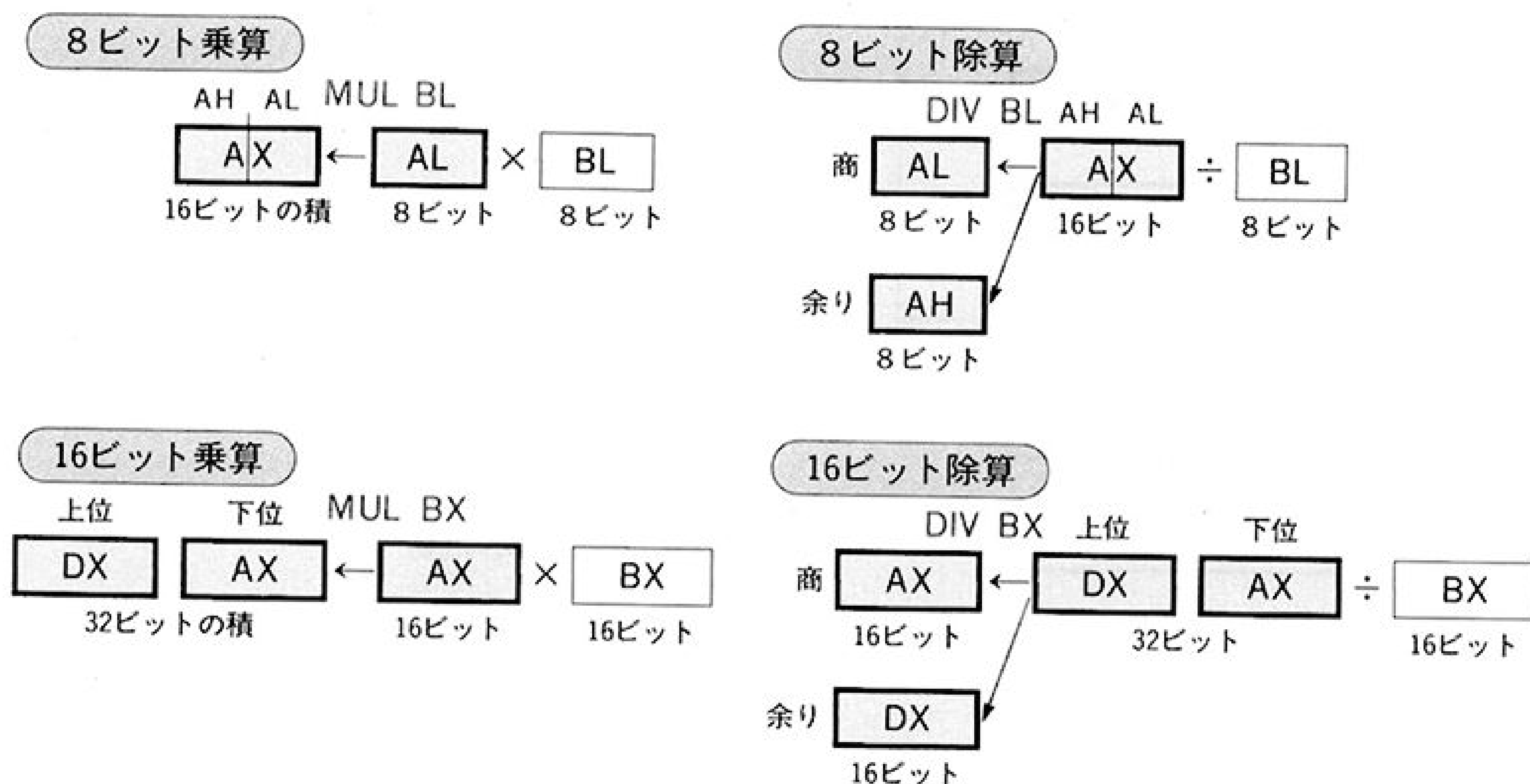


図 6-18　乗除算命令で使用するレジスタ

ここでわかるように乗除算とも，相手となるレジスタが 8 ビットであるか 16 ビットであるかによって，使用するレジスタが異なります.

以下にBL，BXレジスタを例として，乗除算命令の書式をまとめて示します．

＜乗算命令＞

マルチプリケーション(MULtiplication)
MUL　BL ……　ALレジスタの内容（8ビット）にBLレジスタの内容（8ビット）を掛け，結果をAXレジスタ（16ビット）にセットする．

MUL　BX ……　AXレジスタの内容(16ビット)にBXレジスタの内容(16ビット)を掛け，結果をDX:AXレジスタ(32ビット)にセットする．

＜除算命令＞*

ディビィジョン(DIVision)
DIV　BL ……　AXレジスタの内容(16ビット)をBLレジスタの内容(8ビット)で割り，商をALレジスタ（8ビット）に，余りをAHレジスタ（8ビット）にセットする．

DIV　BX ……　DX：AXレジスタの内容(32ビット)をBXレジスタの内容(16ビット)で割り，商をAXレジスタ(16ビット)に，余りをDXレジスタ（16ビット）にセットする．

実習7　乗除算命令

掛け算と割り算の命令の実習を行います．ALレジスタの内容に任意の1バイトデータを掛け，その結果を適当なオフセットアドレスのメモリにストアします．続けてその値を任意の1バイトデータで割り，その結果を次のアドレスにストアしてみましょう．掛け算の結果はAXレジスタに，割り算の結果はALレジスタとAHレジスタにセットされることに注意してください．

最初のALレジスタの内容を「18_H」，掛ける1バイトのデータを「26_H」，割る1バイトのデータを「15_H」としてプログラムを作成すると次のようになります．

* 0による除算など，除算の結果がレジスタに納まらない場合(8ビット←16ビット，16ビット←32ビットであることに注意)は，タイプ0の内部割り込み（214ページ参照）が発生する．

MOV　AL , 18H ……… ALレジスタに1バイトデータ「18_H」をロードする.

MOV　BL , 26H ……… BLレジスタに1バイトデータ「26_H」をロードする.

MUL　BL………………… ALレジスタの内容とBLレジスタの内容を乗算し, 結果をAXレジスタにセットする.

MOV　[0180H], AX… 演算結果をオフセットアドレス「0180_H」にストア.

MOV　BL , 15H ……… BLレジスタに1バイトデータ「15_H」をロードする.

DIV　BL ………………… AXレジスタの内容をBLレジスタで除算し, 商をALレジスタに, 余りをAHレジスタにセットする.

MOV　[0182H], AL… 商をオフセットアドレス「0182_H」にストアする.

MOV　[0183H], AH… 余りをオフセットアドレス「0183_H」にストアする.

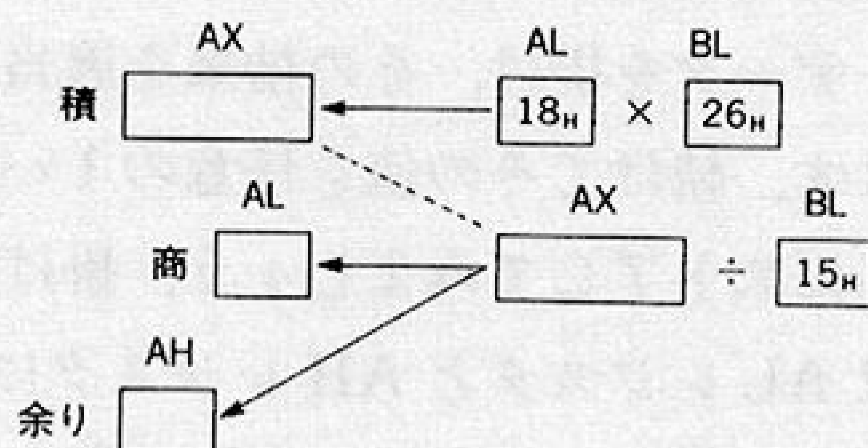
アドレッシングモードの図を見てください. 乗除算命令では演算の対象がALまたはAXレジスタと決まっているので, 矢印の先はそのレジスタだけになっています. また, 矢印の根元がレジスタかメモリしかないことからわかるように, 演算の相手となるのはレジスタかメモリだけです. データそのものを指定することはできないので注意してください.

このプログラムをDEBUG上で入力し, 実行して結果を確かめてみます(図6-19).

```
-A Ø28Ø↵ ……………オフセットアドレス0280Hからアセンブルする
3ØE5:Ø28Ø MOV AL,18↵
3ØE5:Ø282 MOV BL,26↵
3ØE5:Ø284 MUL BL↵
3ØE5:Ø286 MOV [Ø18Ø],AX↵
3ØE5:Ø289 MOV BL,15↵
3ØE5:Ø28B DIV BL↵
3ØE5:Ø28D MOV [Ø182],AL↵
3ØE5:Ø29Ø MOV [Ø183],AH↵
3ØE5:Ø294 ↵
-E Ø18Ø ØØ,ØØ,ØØ,ØØ,ØØ,ØØ,ØØ,ØØ↵ ……………オフセットアドレス0180H付近のメモリを
-D Ø18Ø Ø187↵                             00Hにセットしておく
3ØE5:Ø18Ø  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
-G=Ø28Ø Ø294↵ ……………オフセットアドレス0280Hから0294Hまでのプログラムを実行する

AX=Ø92B  BX=ØØ15  CX=ØØØØ  DX=ØØØØ  SP=FFEE  BP=ØØØØ  SI=ØØØØ  DI=ØØØØ
DS=3ØE5  ES=3ØE5  SS=3ØE5  CS=3ØE5  IP=Ø294   NV UP DI PL NZ NA PE CY
3ØE5:Ø294 E5E5          IN      AX,E5
```

実習7のプログラム



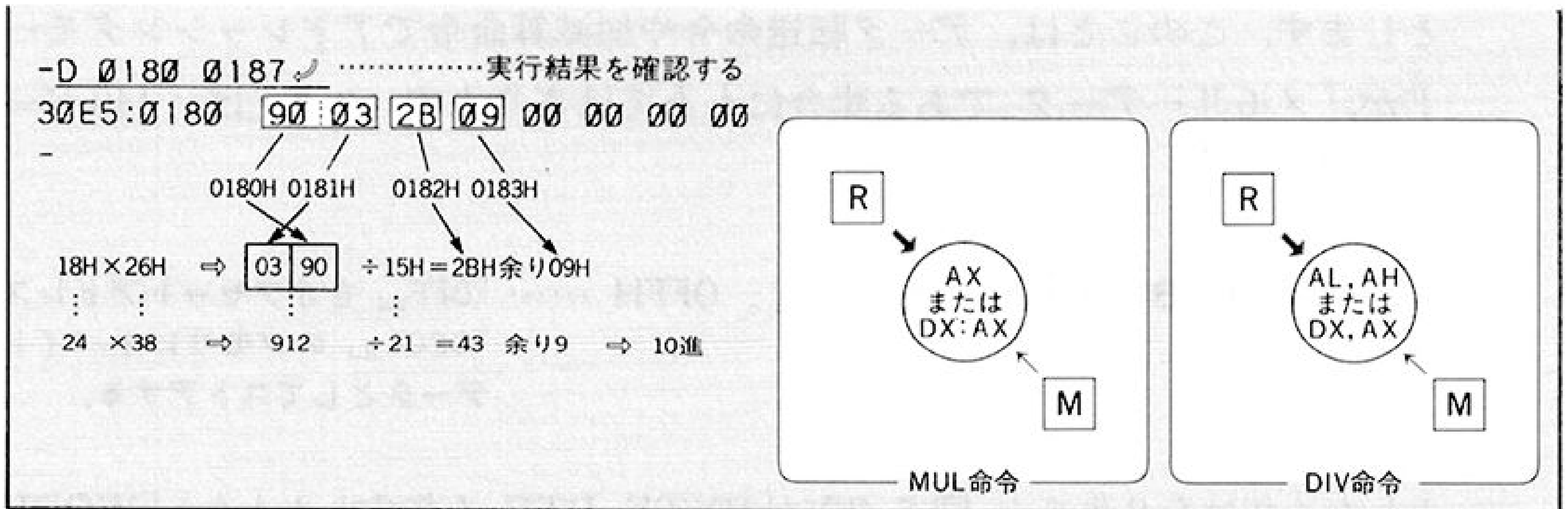


図 6-19　実習 7 の実行結果

PTR 演算子

算術演算命令は，データ転送命令と同じように演算の対象としてメモリの内容をアドレスによって指定することができます．加減算では使用するレジスタによって，1バイトのメモリを対象とするか1ワードのメモリを対象とするかを決めることができます(後で示すように例外もある)．しかし，インクリメント，デクリメント，乗除算などの演算では，メモリを演算の対象とした場合，1バイトのメモリを対象とするのか1ワードのメモリを対象とするのかは単にアドレスを指定しただけではわかりません．

それを指定するためにアセンブリ言語では「PTR 演算子」を使います．PTR 演算子は，

BYTE PTR ［オフセットアドレス］ …… 対象とするメモリは1バイト
WORD PTR ［オフセットアドレス］ …… 対象とするメモリは1ワード

という形式で用います．たとえば，オフセットアドレス「0200_H」の1バイトのメモリをインクリメントする場合には，

INC ［0200H］ → INC BYTE PTR ［0200H］

としなければなりません．また，オフセットアドレス「0200_H」を下位，「0201_H」を上位とする1ワードのメモリをインクリメントする場合には，

INC WORD PTR ［0200H］

とします．このことは，データ転送命令や加減算命令でアドレッシングモードが，「メモリ←データ」である場合にもあてはまります．たとえば，140ページの例は，

MOV　BYTE PTR [0201H] , 0FFH ……「$0FF_H$」をオフセットアドレス「0201_H」のメモリに1バイトデータとしてストアする．

としなくてはなりません．図6-20に「BYTE PTR」を指定したときと「WORD PTR」を指定したときの違いを示します．

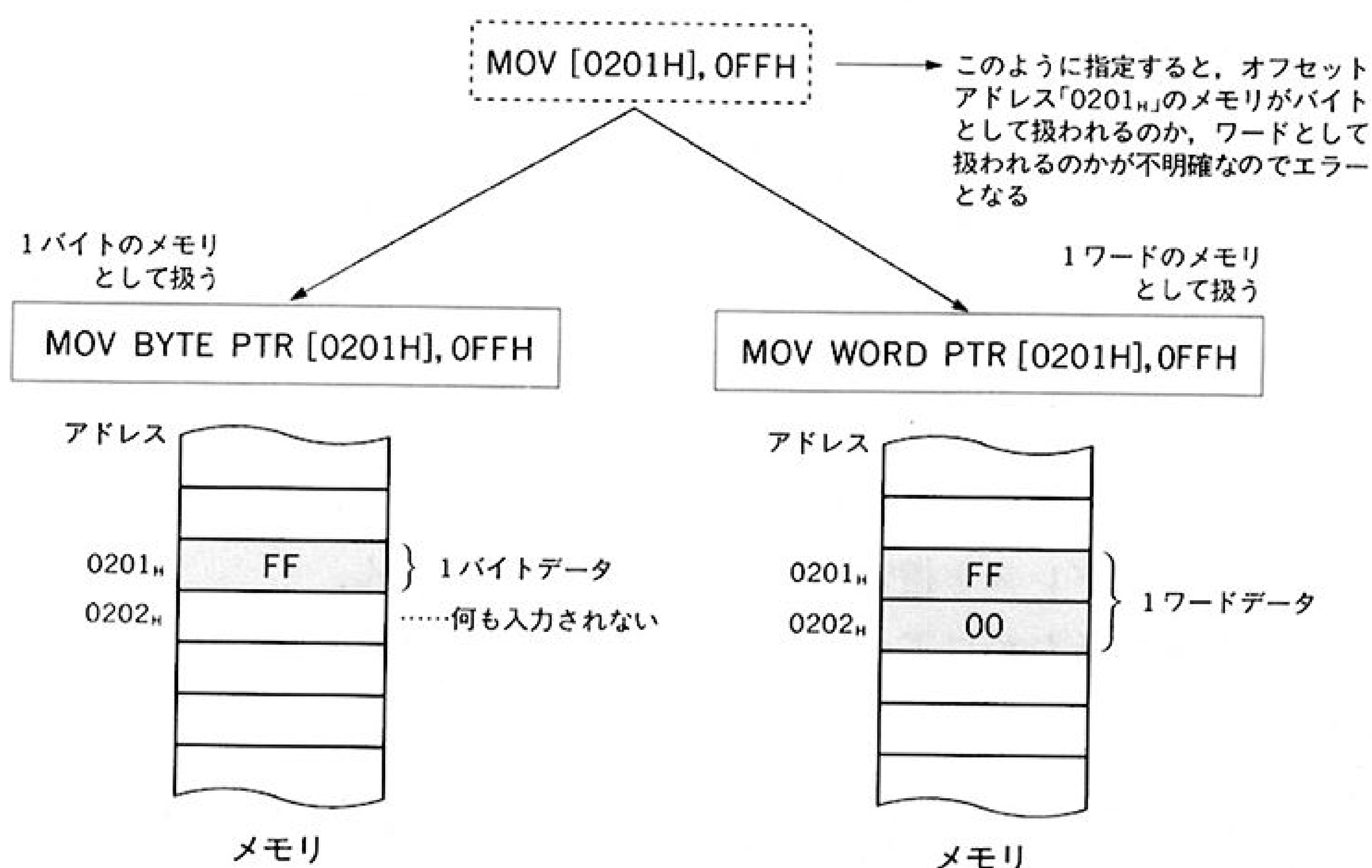


図6-20 「BYTE PTR」と「WORD PTR」

6.4　フラグとジャンプ命令

●

これまでの実習では，プログラムは下位アドレスから上位アドレスへと1命令ずつ実行されてきました．しかし，実行結果によって異なる命令を実行したいときがあります．そのようなときに必要となるのがプログラムの流れを変えるジャンプ命令です．

CPUにフラグがあり，CPUの状態を表すものであることはすでに何度か説明してきました．実は，このフラグは条件ジャンプと密接に関係しています．コンピュータがコンピュータであるための重要な機能の1つに「判断」がありますが，この判断，すなわち「××ならば○○する」という機能こそフラグによる条件ジャンプにほかならないのです．

●

フラグの役割

フラグについては「5.3 レジスタとその機能」でも取り上げていますが，ここでさらにくわしく解説しましょう．

フラグは，前節の算術演算命令などを実行したときに，結果そのものでなく，その結果の状態を保存しておくためにセットされる「旗」のことです．私たちはこの旗の状態を調べることによって，その後の行動を決定しプログラムを進行させます．

フラグは次の**表6-3**にあるように，フラグレジスタのそれぞれ該当するビットとして示されます．フラグレジスタは他のレジスタのように8ビットや16ビットという単位で動作するのではなく，SF（サインフラグ），ZF（ゼロフラグ），CF（キャリーフラグ）などの個々のフラグごと，つまり1ビット単位で単独に動作するのです．そのためフラグレジスタを，他のレジスタのように16ビットのレジスタとしてプログラムから自由に操作することはできません．また，それぞれのフラグの状態を言い表す場合，1にすることを「セッ

ト」，0にすることを「リセット」（クリア）と呼びます．

フラグを変化させるのは主に算術演算命令や6.7節で取り上げる論理演算命令ですが，命令ごとに影響を受けるフラグは決まっています．各命令の実行で影響を受けるフラグはAPPENDIXの命令一覧表にまとめていますから，参照してください．

フラグレジスタの各フラグの機能を次の**表6-3**に示しておきますが，すべてのフラグの名前や機能を覚える必要はありません．必要に応じて覚えていけばよいでしょう．この節ではこれらのフラグの中からゼロフラグを使って実習を行うことにします．

〈フラグレジスタ〉

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0 (ビット)
＼	＼	＼	＼	OF	DF	IF	TF	SF	ZF	＼	AF	＼	PF	＼	CF

↑使用されていないフラグ

フラグ	名称	機能	DEBUGでの表示	
			1のとき	0のとき
OF	オーバーフローフラグ	符号付き演算で桁あふれが生じたときにセットされる	OV	NV
DF	ディレクションフラグ	ストリング操作命令においてポインタの増減方向を示す	DN	UP
IF	インタラプト・イネーブルフラグ	クリアすると外部割り込みを受け付けなくなる	EI	DI
TF	トラップフラグ	シングルステップモード*で実行するときのフラグ	—	—
SF	サインフラグ	演算結果の符号を表す．負ならばセットされる	NG	PL
ZF	ゼロフラグ	演算結果がゼロであればセットされる	ZR	NZ
AF	補助キャリーフラグ	BCD演算で使用されるキャリーフラグ	AC	NA
PF	パリティフラグ	演算の結果，1となるビット数が偶数のときセットされ，奇数のときリセットされる	PE	PO
CF	キャリーフラグ	演算の結果，桁上がりが生じた場合にセットされる	CY	NC

＊DEBUGのTコマンドなどで使用される．

〈DEBUGのRコマンドでの表示〉

```
-R↵
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3528  ES=3528  SS=3528  CS=3528  IP=0100   NV UP EI PL NZ NA PO NC
3528:0100 0000          ADD     [BX+SI],AL
```

1つ1つがフラグの状態を表している

表6-3　フラグレジスタの中身とその機能

プログラムの実行順序を変更する命令

プログラムは，メモリのアドレスの低い方から高い方へ向かって順に格納されており，CPU はそれを IP（インストラクションポインタ）レジスタの指しているところから取り出して解析し，実行していきます．命令が実行されるとともに，IP レジスタは次の命令を指すように自動的にインクリメントされていきます．このことは，「5.6 プログラム実行のメカニズム」でくわしく解説しました．

つまり，プログラムの流れは IP レジスタがコントロールしており，この IP レジスタを強制的に操作しない限り，プログラムはアドレスの下位から上位へと順次進んでいくだけです．

IP レジスタは，他のレジスタとは異なり MOV 命令で値をロードすることはできません．IP レジスタを書き換えることはプログラムの実行の流れを変えることになるため，それ専用の特別な命令が用意されているのです．IP レジスタを強制的に操作して，プログラムの流れを変える命令には次の 4 つがあります．

①ジャンプ命令
②コール命令
③リターン命令
④割り込み命令（これはプログラム上の命令でもあるが，本来は CPU をハードウェア的にコントロールする方法）

いずれもコンピュータのプログラミング上なくてはならない重要な命令であり，これらがなくてはほとんどのプログラムは作ることができないでしょう．

本節では①のジャンプ命令の実習解説を行います．②のコール命令と③のリターン命令については次の 6.5 節で実習解説を行います．④の割り込み命令については 6.10 節で解説します．

フラグと条件ジャンプ

ジャンプ命令には「無条件ジャンプ命令」と「条件ジャンプ命令」の2つがあります．ジャンプ命令を実行すると，IPレジスタには次の命令のアドレスではなく，ジャンプ命令で指定したアドレス値がロードされます．それまでは次の命令へ次の命令へと順次プログラムが実行されていたのに対し，ジャンプ命令を実行するとジャンプ命令の次の命令ではなくジャンプ命令で指定したアドレスへと飛んでいく，つまり「ジャンプ」するのです．

無条件ジャンプ命令では，どんな場合でも必ずIPレジスタに指定されたアドレスがロードされ「ジャンプ」しますが，条件ジャンプ命令では，ある条件のときにだけしかジャンプしません．その条件が満たされない場合は，IPレジスタにはジャンプ命令で指定されたアドレス値はロードされず，単にジャンプ命令の次の命令を指すようにインクリメントされるだけです．この条件ジャンプによって，「××ならば○○する」という判断が可能になります．

先に「フラグ」について解説しましたが，フラグはこの条件ジャンプ命令で利用するために存在するものです．つまり，ある条件によってプログラムの進行を制御する命令が条件ジャンプ命令であり，条件を判断する材料がフラグであるわけです．そして，そのフラグをセットするための命令が前節で解説したコンペア（CMP）命令です*．

たとえば，ALレジスタにキーボードから入力された文字が入っており，それがリターンキーによる文字（キャラクタコード「$0D_H$**」）に等しいかどうかを知りたければ（つまりリターンキーが押されたかどうかを調べたければ），「CMP AL,0DH」という命令を実行します．コンペア命令によってALレジスタの内容と1バイトデータ「$0D_H$」とが比較され，同じ値ならばZF（ゼロフラグ）が「1」にセットされ，そうでなければ「0」にリセットされます．

このようにコンペア命令を実行すれば，ALレジスタの内容が「$0D_H$」かどうかがわかるのです（もちろん，減算命令「SUB」を使ってもフラグの変化は同じなので判断することができます）．また，ALレジスタの内容の方が小さければCF（キャリーフラグ）が「1」にセットされ，そうでなければ「0」

---

*算術演算命令を使ってもフラグはセットされる．

**キーボードの各キーは対応する文字のキャラクタコードで表される．リターンキーなどは，2章26ページで「表示できない文字」と言っている文字に対応している（273ページのキャラクタコード表を参照）．

にリセットされます．このことによってALレジスタの内容が「$0D_H$」より小さいかどうかがわかります．

とはいっても，コンペア命令を実行しただけでは，その結果にしたがってフラグがセットされているだけで，私たちにはまだわかっていません．その次の命令で必要なフラグを見に行って，初めてその結果がわかるのです．

必要なフラグを見に行き，その結果によりプログラムの流れを変えるのが条件ジャンプ命令です．つまり条件ジャンプ命令は「フラグを見に行く」仕事と，「それによりプログラムの流れを変えたり変えなかったりする」仕事の2つを同時に行うのです．

上の例では，ALレジスタの内容が「$0D_H$」に等しいときにだけ，つまりリターンキーが押されたときだけなんらかの処理をさせたければ，コンペア命令を実行したあと，「ZF(ゼロフラグ)を見に行って1にセットされていればジャンプする」という条件ジャンプ命令を実行すればよいのです．これをプログラムで表すと次のようになります（図6-21）．

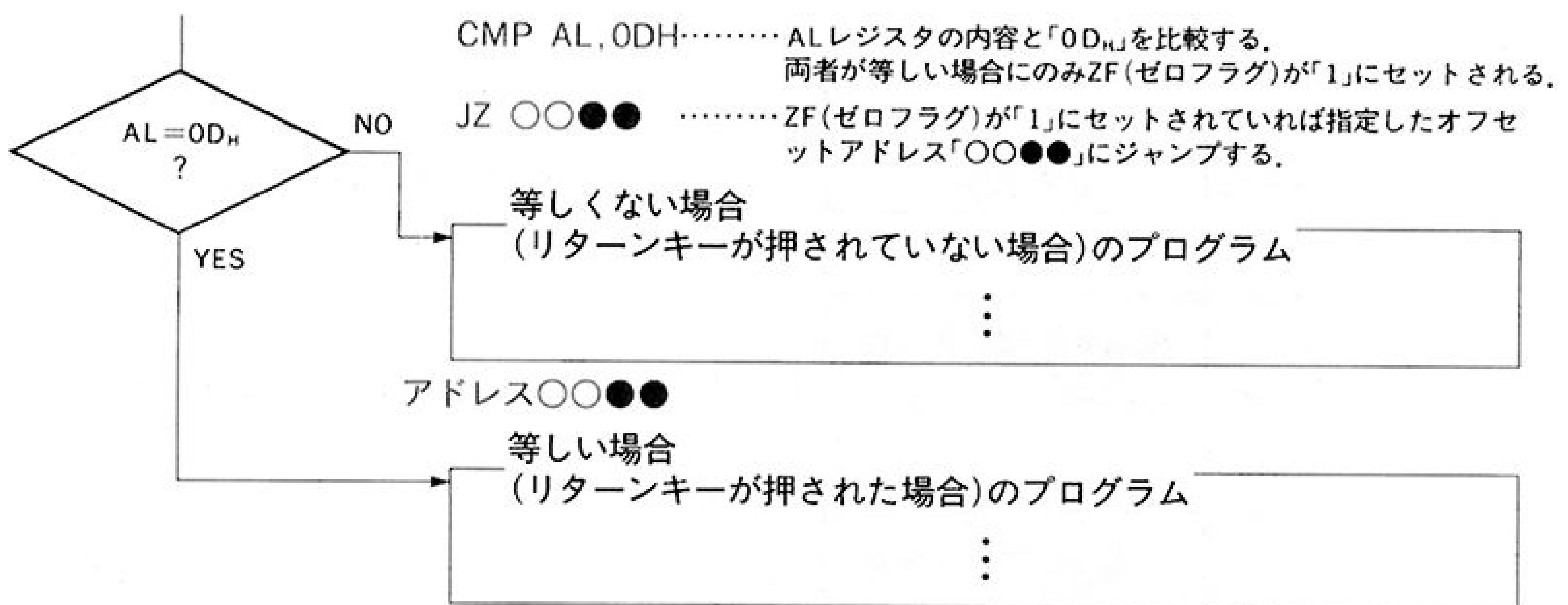


図6-21　ゼロフラグによる条件ジャンプ

ここではゼロフラグを利用した条件ジャンプの一例を示しました．この例から「フラグ」と「条件ジャンプ」の概念が一応理解できるのではないかと思います．

実習 8　無条件ジャンプ

まず，ジャンプ命令によってIPレジスタにそのジャンプ命令で指定されているアドレス値がロードされ，そのアドレスからプログラムが実行されることを確認しましょう．そのために次の無条件ジャンプ命令を実習します．

無条件ジャンプ命令は，

ジャンプ(JuMP)
JMP ○○●● …… オフセットアドレス「○○●●」からのプログラムを実行する．

という形式で表します．この命令が実行されると，IPレジスタに指定されたオフセットアドレスがロードされ，「ジャンプ」が実行されます．

実習6（プログラムのスタートアドレス＝0200_H）および実習7（プログラムのスタートアドレス＝0280_H）のプログラムを利用してジャンプ命令を実習しましょう．

この一連のプログラムは，次の図6-22のようにメモリ上に配置されます．

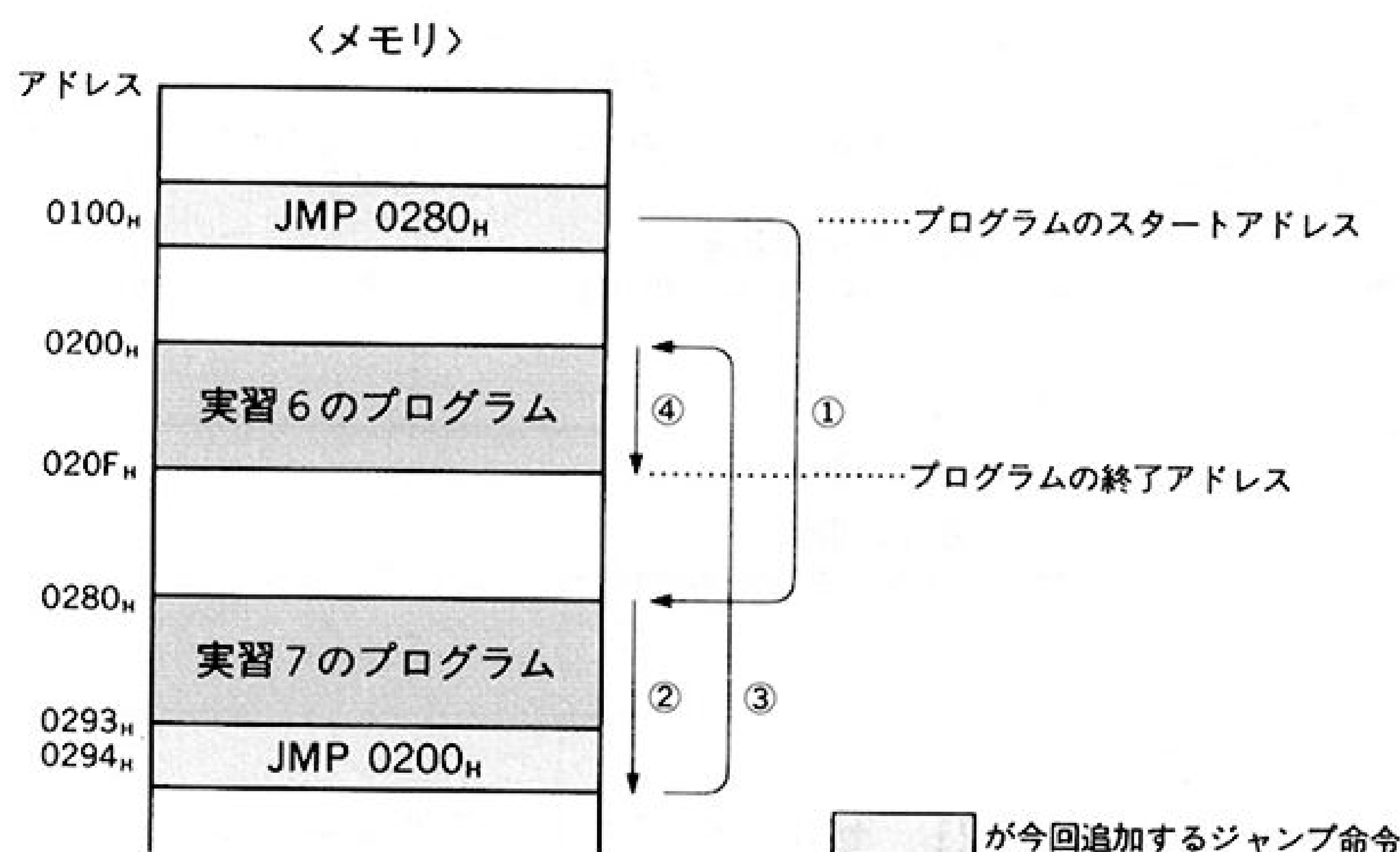


図6-22　実習6と実習7のプログラムの配置

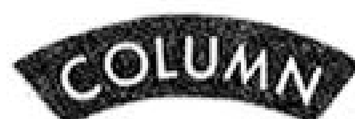
この図で示すように，実習6，7のプログラムに無条件ジャンプ命令を追加してプログラムを作ってみます．DEBUG上で入力し，実行して結果を確認してみましょう（図6-23）．

```
-A 0200         ……オフセットアドレス0200Hからアセンブルする
3717:0200 MOV BL,10
3717:0202 MOV AL,BL
3717:0204 DEC AL
3717:0206 MOV [0170],AL        実習6のプログラム
3717:0209 INC BL
3717:020B MOV [0171],BL
3717:020F
-A 0280         ……オフセットアドレス0280Hからアセンブルする
3717:0280 MOV AL,18
3717:0282 MOV BL,26
3717:0284 MUL BL
3717:0286 MOV [0180],AX        実習7のプログラム
3717:0289 MOV BL,15
3717:028B DIV BL
3717:028D MOV [0182],AL
3717:0290 MOV [0183],AH
3717:0294 JMP 0200   ……オフセットアドレス0200H(実習6)へのジャンプ命令
3717:0297
-A 0100         ……オフセットアドレス0100Hからアセンブルする
3717:0100 JMP 0280   ……オフセットアドレス0280H(実習7)へのジャンプ命令
3717:0103              (プログラムのスタートアドレス)
-E 0170 00,00,00,00,00,00,00,00  } オフセットアドレス0170Hと0180H付近の
-E 0180 00,00,00,00,00,00,00,00  } メモリを00Hにセット
-D 0170 018F
3717:0170  00 00 00 00 00 00 00 00-06 75 13 BA 01 00 52 8B   .........u.:..R.
3717:0180  00 00 00 00 00 00 00 00-18 0E 5B 5B 2B F6 83 FF   ..........[[+v..
-G=0100 020F    ……オフセットアドレス0100Hからのプログラムを実行
        └→ 実習6のプログラムの終了アドレスである点に注意(図6-22参照)
AX=090F  BX=0011  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3717  ES=3717  SS=3717  CS=3717  IP=020F   NV UP EI PL NZ NA PE CY
3717:020F E5E5          IN      AX,E5
-D 0170 018F    ……実行結果を確認する
3717:0170  0F 11 00 00 00 00 00 00-06 75 13 BA 01 00 52 8B   .........u.:..R.
3717:0180  90 03 2B 09 00 00 00 00-18 0E 5B 5B 2B F6 83 FF   ..+.......[[+v..
-
        └→ 実習6,7と同じ結果が得られた
```

DEBUGを終了せずに，実習6，7と続けて実習している場合は入力済みなので不要

-A 0294としてここから始めればよい

図6-23　実習8のプログラムと実行結果



ジャンプ先のアドレスがマシン語に変換されると…

Aコマンドでアセンブリ言語のニーモニックを入力するときに,

JMP 0280

のようにジャンプする先のオフセットアドレスを入力しますが，これがどのようなマシン語に変換されているかに注目してください．データ転送命令でデータをメモリに転送する場合には,

MOV AX , [0156] → A1 56 01
逆アセンブル

というマシン語に変換されました．オフセットアドレスは1ワードデータですから，その上位バイトと下位バイトが逆になった形で格納されています．これに対し，実習9のジャンプ命令は,

JMP 0280 → E9 7D 01
逆アセンブル

というマシン語に変換されています（実際に逆アセンブルして確かめてみてください)．オフセットアドレスの値が見あたらず，その代わり「$017D_H$」という1ワードデータに変換されています（上位バイト，下位バイトが逆転していることに注意)．これはどういうことかというと，ジャンプ命令の飛び先となるオフセットアドレスは，オフセットアドレスそのままの値ではなく，そのジャンプ命令自身の次の命令からジャンプ先までの距離を示すバイト数で表されているのです．この場合，ジャンプ命令の次のオフセットアドレスは「0103_H」で，これにそのバイト数「$017D_H$」を加えると「0280_H」となりジャンプ先のオフセットアドレスとなります．

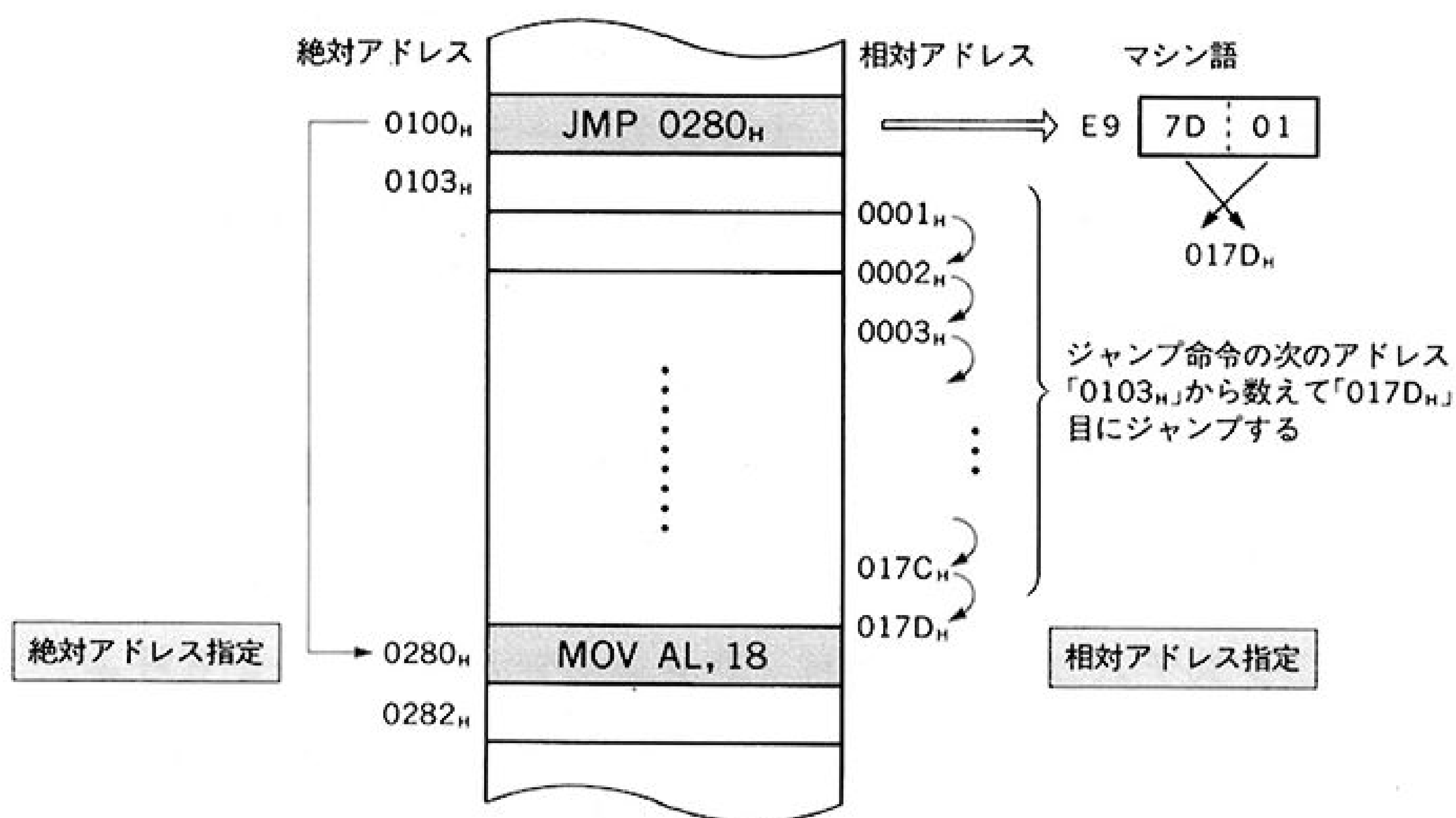


図　相対アドレス指定と絶対アドレス指定

このようなアドレスの指定方法を，**相対アドレス**指定と呼びます．8086CPUのジャンプ命令は，すべてこの相対アドレスによってジャンプ先を指定します．マシン語は相対アドレスによって実行されますが，アセンブリ言語では飛び先のオフセットアドレスそのもの*（**絶対アドレス**と言う）を指定するので注意が必要です．

* 8章で紹介するMASMでは，オフセットアドレスではなくラベルを使って指定する．

実習9　条件ジャンプ(1)

コンペア命令の実習をここで条件ジャンプに関係づけて実習解説します．

ここで取り上げる例題は，前項の「条件ジャンプ」のところで説明した「リターンキーが押されたか」という問題を考えてみましょう．

この項からは，コンソール，すなわちキーボードやスクリーンとの入出力や，プログラムを終了してDEBUGに戻るためなどにMS-DOSのシステムコールを利用します．システムコールについては6.10節でくわしく解説しますので，今のところは次のような命令を実行すればこういうことができる，ということだけを理解しておいてください．

ここで利用するシステムコールは次の2つです．

◇システムコール　21H（ファンクションコールとも呼ぶ）

・ファンクション1番　……　**コンソールから1文字入力し，ALレジスタに格納する．**

（使用例）MOV　AH，1
INT（インタラプト (INTerrupt)）　21H

・ファンクション9番　……　**DXレジスタで指定されるアドレスから始まる文字列をコンソールに出力する．**

（使用例）MOV　AH，9
MOV　DX，○○●●　…1ワードのオフセットアドレス値
INT　21H

◇システムコール　20H

・プログラムを終了してDEBUGに戻る．

（使用例）INT　20H

ファンクション1番の2命令を実行すると，キー入力待ちになり，キーが押されるとそのキャラクタコードがALレジスタに格納されます．ファンクション9番の3命令を実行すると，アドレス「○○●●」から始まるメモリに格納されている文字列がコンソール（画面）に出力されます．

プログラムでは，まずファンクション1番を使ってキーボードから1文字

入力します．キーボードを押すとそのキーのキャラクタコードがALレジスタに格納されるので，コンペア命令でリターンキーのキャラクタコード「$0D_H$」と比較します．条件ジャンプ命令で，ZF（ゼロフラグ）を調べ，1にセットされていればジャンプします．ジャンプした先ではファンクション9番を使って「リターンキーが押されました」と表示します．ZFがセットされていなければ，プログラムはそのまま実行を続け，そこではファンクション9番を使って「リターンキーが押されませんでした」と表示します．

ZFがセットされていればジャンプするという条件ジャンプ命令は，

ジャンプイフゼロ(Jump if Zero)
JZ　○○●●　……　ZFがセットされていれば，オフセットアドレス「○○●●」からのプログラムを実行する．

という形式で表します．

「実習9」のプログラムは以下のようになります．

```
        MOV AH,1 … ファンクションコール1番(コンソールから1文字入力する).
        INT  21H
        CMP  AL , 0DH … ALレジスタの内容と1バイトデータ「0DH」とを比較
                        する.
 ┌──── JZ [リターンキー] … ゼロフラグが「1」であれば，[リターンキー]へジャンプ
 │                          する.
 │      MOV  AH  , 9        ┐ファンクションコール9番(コンソールに文字列
 │      MOV  DX  , ○○●● H ┤を出力する). オフセットアドレス「○○●●H」
 │      INT  21H            │には「リターンキーが押されませんでした」とい
 │                          ┘う文字列をセットしておく.
 │ ┌── JMP [終了]      ……………… [終了]へ無条件ジャンプする.
 └─┼→[リターンキー]
   │    MOV  AH  , 9        ┐ファンクションコール9番(コンソールに文字列
   │    MOV  DX  , ●●○○ H ┤を出力する). オフセットアドレス「●●○○H」
   │    INT  21H            │には「リターンキーが押されました」という文字
   │                        ┘列をセットしておく.
   └→[終了]
        INT  20H ………… プログラムを終了しDEBUGに戻る.
```

プログラムを実行する前に，あらかじめDEBUGのEコマンドで出力する文字列をメモリに書き込んでおきます．このプログラムをDEBUG上で入力し，実行して結果を確かめてみましょう．

```
-A Ø1ØØ↵ ……………オフセットアドレス0100Hからアセンブルする
3A9F:Ø1ØØ MOV AH,1↵
3A9F:Ø1Ø2 INT 21↵
3A9F:Ø1Ø4 CMP AL,ØD↵
3A9F:Ø1Ø6 JZ Ø111↵
3A9F:Ø1Ø8 MOV AH,9↵
3A9F:Ø1ØA MOV DX,Ø18Ø↵
3A9F:Ø1ØD INT 21↵
3A9F:Ø1ØF JMP Ø118↵
3A9F:Ø111 MOV AH,9↵
3A9F:Ø113 MOV DX,Ø1CØ↵
3A9F:Ø116 INT 21↵
3A9F:Ø118 INT 2Ø↵
3A9F:Ø11A ↵
-E Ø18Ø 'リターンキーが押されませんでした.',ØD,ØA,'$'↵
-E Ø1CØ 'リターンキーが押されました.',ØD,ØA,'$'↵
-G=Ø1ØØ↵ …オフセットアドレス0100Hからプログラムを実行する
Aリターンキーが押されませんでした.
「A」のキーを押した
Program terminated normally
-G=Ø1ØØ↵ ……………再び実行する
リターンキーが押されました.
↵キーを押した
Program terminated normally
-
```


図6-24　実習9の実行結果

●

各種の条件ジャンプ命令

「実習9」では，コンペア命令を実行した後，条件ジャンプ命令によってZF（ゼロフラグ）が「1」にセットされているかどうかを調べてジャンプしました．つまり，比較の結果が等しいかどうかを調べたのです．

これ以外にも，たとえば実習9の「CMP　AL,0DH」という命令の後ならば，次のような条件ジャンプ命令を使うことができます（表6-4）．

命　　令	機　　能
Jump if Zero JZ　○○●●	(AL＝$0D_H$)ならばジャンプする
Jump if Not Zero JNZ ○○●●	(AL≒$0D_H$)ならばジャンプする
Jump if Below JB　○○●●	(AL＜$0D_H$)ならばジャンプする
Jump if Above JA　○○●●	(AL＞$0D_H$)ならばジャンプする
Jump if Below or Equal JBE ○○●●	(AL≦$0D_H$)ならばジャンプする
Jump if Above or Equal JAE ○○●●	(AL≧$0D_H$)ならばジャンプする

表 6-4　各種の条件ジャンプ命令

この表で示した条件ジャンプ命令は，比較する 2 つの値(この場合は AL レジスタと $0D_H$)の，どのような大小関係を調べたいかによって使い分ければよいのです*.

上に挙げた命令は，比較した 2 つの値をそれぞれ符号なしの 1 バイトデータとみなした場合（10 進では 0 ～255 の値）の命令です．これに対して，1 バイトのデータを符号ありと考えた場合（10 進では－128～＋127 の値）の条件ジャンプ命令も用意されています．それを以下に挙げておきます(**表 6-5**).

命　　令	機　　能
Jump if Less JL　○○●●	(AL＜$0D_H$)ならばジャンプする
Jump if Greater JG　○○●●	(AL＞$0D_H$)ならばジャンプする
Jump if Less or Equal JLE ○○●●	(AL≦$0D_H$)ならばジャンプする
Jump if Greater or Equal JGE ○○●●	(AL≧$0D_H$)ならばジャンプする

表 6-5　負の数を扱う場合の条件ジャンプ命令

＊これらの条件ジャンプ命令は，実際には ZF（ゼロフラグ），CF（キャリーフラグ），SF（サインフラグ），OF（オーバーフローフラグ）などを CPU が調べ，その状態によってジャンプするかどうかが判断される．なお，APPENDIX の命令一覧では，それぞれの条件ジャンプ命令の実行に関係するフラグとその条件を示す．

データを符号ありとした場合と，符号なしとした場合の扱いについては4.2章で解説しているのでもう一度参照してください．具体的にどういう違いがあるのか，例を使って説明しましょう．

先のコンペア命令「CMP AL,0DH」においてALレジスタの内容が「FF_H」であったとします．データを符号なしとして扱った場合にはALレジスタの内容「FF_H」は10進では255，「$0D_H$」は13となります．この2つを比較した結果は，

AL　＞　$0D_H$
255　　13　… 10進

となります．データを符号なしとして扱う場合には，条件判定には**表6-4**の条件ジャンプ命令グループを使います．したがってJB, JBE命令ではジャンプせず，JA，JAE命令ではジャンプします．

ところがデータを符号ありとして扱う場合には，ALレジスタの内容「FF_H」は10進では−1，「$0D_H$」は同様に13となります．この2つを比較した結果は，

AL　＜　$0D_H$
−1　　13　… 10進

となり，さきほどとは異なる結果になります．データを符号ありとして扱う場合には，条件判定には**表6-5**の条件ジャンプ命令グループを使います．したがってJL, JLE命令ではジャンプし，JG, JGE命令ではジャンプしません．

このことをわかりやすく図解したのが次の**図6-25**です．この図は，「CMP AL,0DH」という命令を実行したときに，各ジャンプ命令でALレジスタの内容がどの範囲にあるときにジャンプするのかを示したものです．



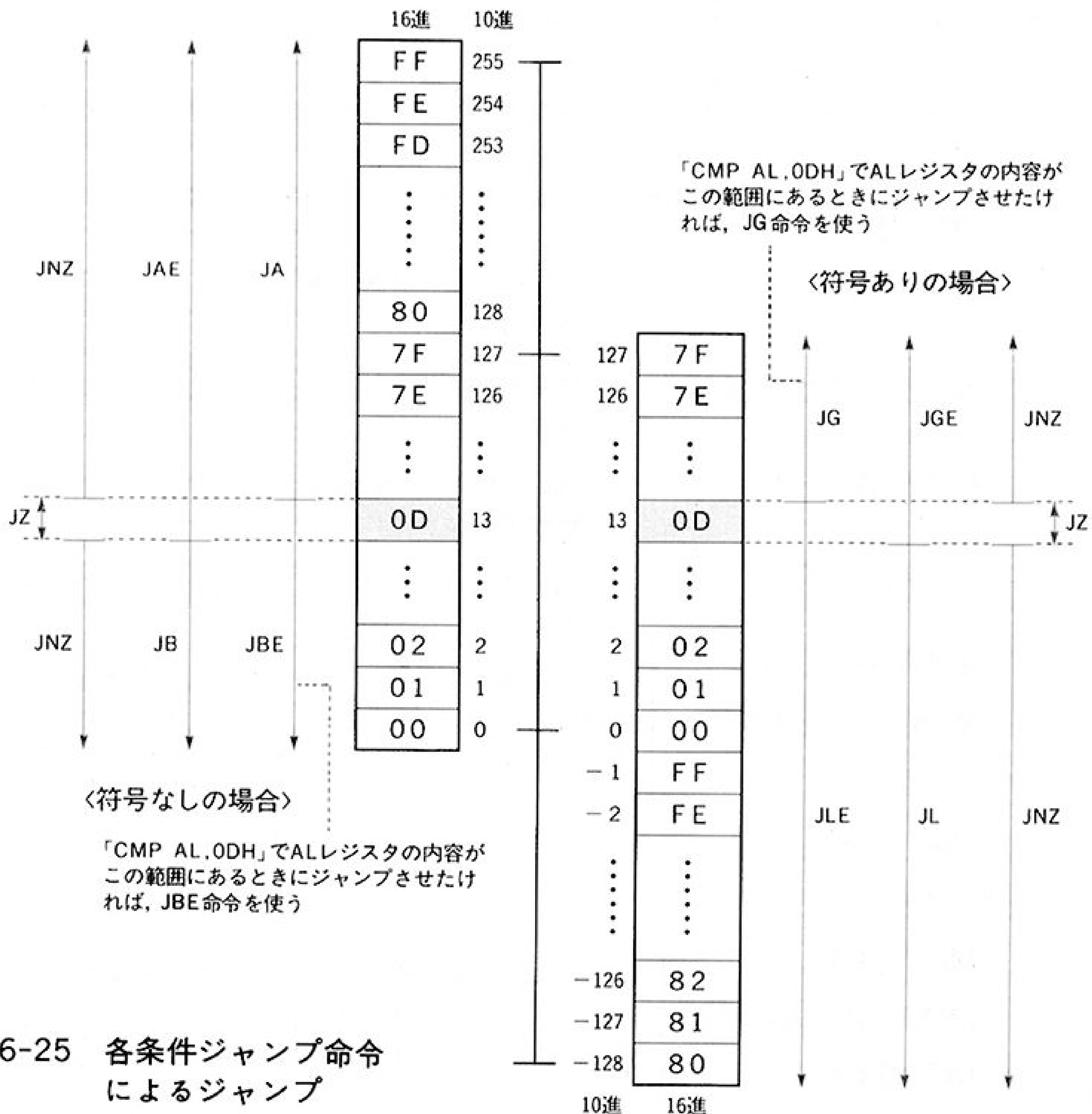


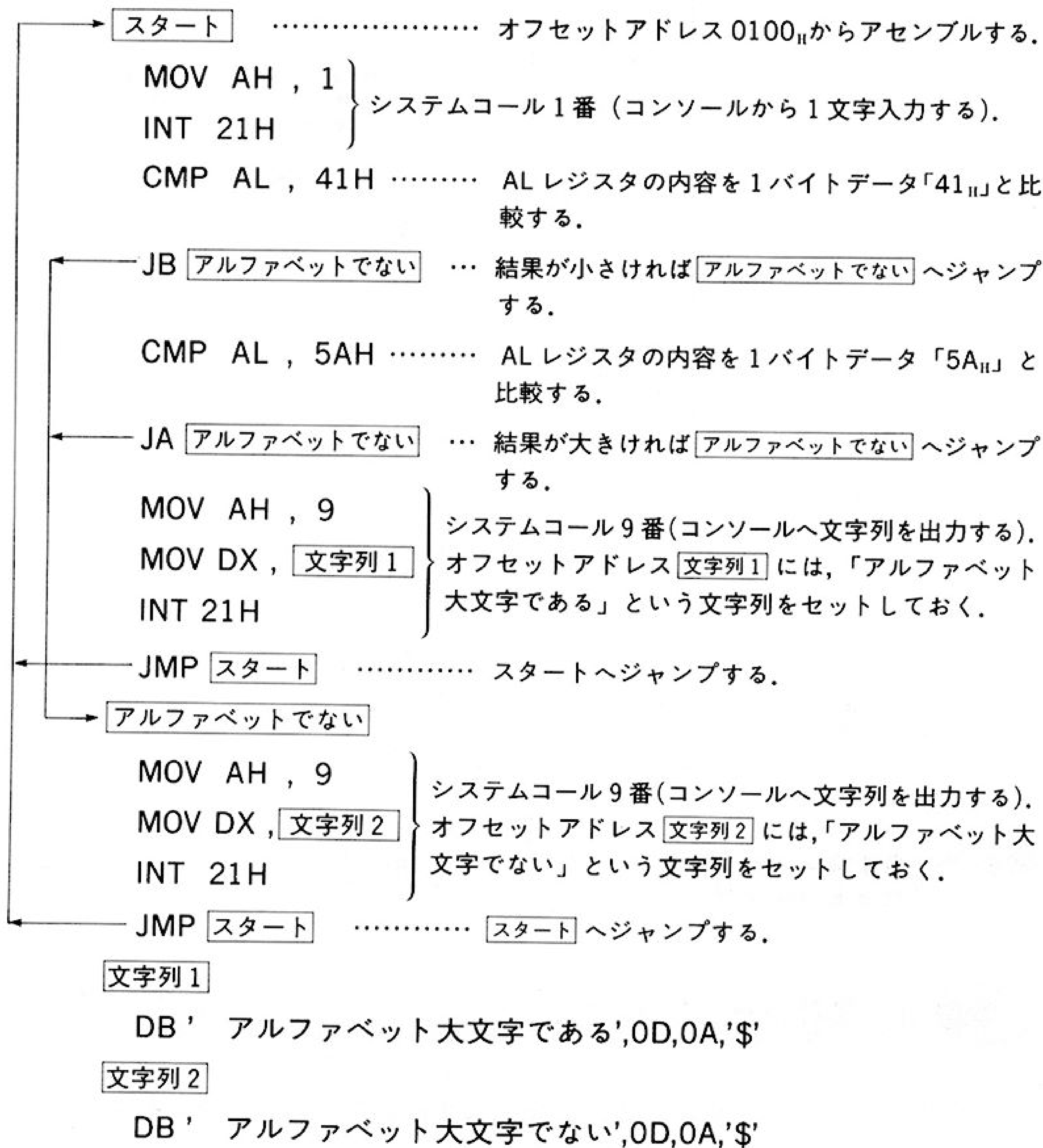
図6-25　各条件ジャンプ命令によるジャンプ

実習10　条件ジャンプ（2）

条件ジャンプ命令を利用して，数値の大小関係を比較し，その結果に応じた処理をするプログラムを実習しましょう．

キーボードから1文字入力し，入力した文字が大文字かどうかによって対応するメッセージを表示するプログラムを作ります．アルファベット大文字のキャラクタコードは「41_H」から始まり，ABC順に連続しています(APPENDIXのキャラクタコード表を参照)．「A」のキャラクタコードは「41_H」で，「Z」のキャラクタコードは「$5A_H$」ですから，入力した文字の値が41_H以上$5A_H$以下であれば，アルファベット大文字であるとみなすことができます．

実習10のプログラムは次のように作成しましょう.



表示する文字列は，これまではEコマンドによってメモリにセットしてきましたが，ここではデータ定義命令を用いてプログラムの一部としてセットします.

その書式は,

DB データ列

└── Define Byte（バイト列を定義する）

となり，DEBUGのEコマンドと同様にデータ列には「00，01，02……」とカンマで区切ってデータを並べます．また，シングルクォーテーション（'）でデータを囲むと該当するキャラクタコードに変換されて入力されます．

注意して欲しいのは，この「DB」という命令はマシン語に対応するニーモニックではなくアセンブリ言語の「擬似命令*」であることです．つまりCPUが直接実行するマシン語命令ではなく，データをメモリ上にセットするための指示であると思ってもらえばよいでしょう．したがって逆アセンブルしてもDBという命令が出てくるわけではありません．

プログラムをDEBUGで入力し，実行した結果を次の図6-26に示しましょう．このプログラムは無限ループになっていますから，CTRL+C を入力して終了してください．

```
-A 0100 ↲ ············オフセットアドレス0100Hからアセンブルする
3A9F:0100 MOV AH,1↲
3A9F:0102 INT 21↲
3A9F:0104 CMP AL,41↲
3A9F:0106 JB 0115↲
3A9F:0108 CMP AL,5A↲
3A9F:010A JA 0115↲
3A9F:010C MOV AH,9↲
3A9F:010E MOV DX,011E↲
3A9F:0111 INT 21↲
3A9F:0113 JMP 0100↲
3A9F:0115 MOV AH,9↲
3A9F:0117 MOV DX,013D↲
3A9F:011A INT 21↲
3A9F:011C JMP 0100↲
3A9F:011E DB '　アルファベット大文字である',0D,0A,'$'↲
3A9F:013D DB '　アルファベット大文字でない',0D,0A,'$'↲
3A9F:015C ↲
-G=0100 ↲ ··············オフセットアドレス0100Hからプログラムを実行する
C　アルファベット大文字である
3　アルファベット大文字でない
d　アルファベット大文字でない
^C ·············CTRL+C でプログラムを終了する
AX=0124  BX=0000  CX=0000  DX=013D  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3A9F  ES=3A9F  SS=3A9F  CS=3A9F  IP=0104   NV UP EI PL NZ AC PE NC
3A9F:0104 E5E5          IN      AX,E5
-
```

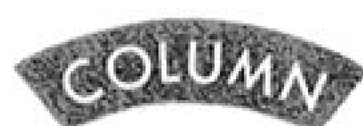
実習10のプログラム
〈注意〉
実習9と同様にジャンプ先のアドレスなどは本来わからないはずであるが，実習用にあらかじめ調べてある

マシン語命令でなく，データをセットする命令であることに注意

C，3，d の順にキーを押した

図6-26　実習10の実行結果

*擬似命令については8.2章で解説する．



MS-DOS の実行型ファイルを作成するには……

これまで実習してきたプログラムは，DEBUG コマンドを抜けて MS-DOS に戻るとすべて消えてなくなってしまいます．しかし，実習 10 のようなプログラムは単に DEBUG 上で実行するだけでなく，ディスク上のプログラムとして残しておきたいという人も多いでしょう．実は，実習のプログラムは，ちょっと手を加えるだけで簡単に実行型ファイルとしてとっておくことができます．そこで以下にその基本的な手順を示しておきましょう．

①Rコマンドを使って，BX レジスタに 0 を，CX レジスタにプログラムサイズ（<プログラムの終了アドレス>－0100_H）をセットする*.

②N（Name）コマンドを使って，実行形式のファイル名（.COM）をセットする．

③Wコマンドを使って，ディスクに書き出す．

このとき注意することは，プログラムのスタートアドレスが必ず「0100_H」でなければならないことです．そうでない場合はプログラムを変更してください．また，プログラムはシステムコール 20_H，つまり，

INT 20H

という命令で終了していなければなりません**.

それでは実際に，実習 10 のプログラムを使って MS-DOS の実行型ファイルを作成してみましょう．

```
-A 0100↵          ─→ プログラムのスタートアドレス
3A9F:0100 MOV AH,1
3A9F:0102 INT 21
3A9F:0104 CMP AL,41
3A9F:0106 JB 0115
```

＊プログラムがデータを含む場合は，データも含めたプログラムのサイズをセットする．

＊＊実習 10 のプログラムのように [CTRL]+[C] で中止しない限り，永遠に同じことを繰り返すようなものは該当しない．

```
3A9F:0108 CMP AL,5A
3A9F:010A JA 0115
3A9F:010C MOV AH,9
3A9F:010E MOV DX,011E
3A9F:0111 INT 21
3A9F:0113 JMP 0100
3A9F:0115 MOV AH,9
3A9F:0117 MOV DX,013D
3A9F:011A INT 21
3A9F:011C JMP 0100
3A9F:011E DB '  アルファベット大文字である',0D,0A,'$'
3A9F:013D DB '  アルファベット大文字でない',0D,0A,'$'
3A9F:015C ………プログラムの終了アドレス
-R BX↵ ……………BXレジスタの内容を確認する
BX 0000……………現在のBXレジスタの値
:0000↵ ……………BXレジスタを「0」にセットする(この場合，現在のBXレジスタの値が0なので単に
                                     リターンキーを押してもよい)
-R CX……………CXレジスタの内容を変更する
CX 0000……………現在のCXレジスタの値
:005C↵ ……………CXレジスタにプログラムサイズをセットする(<終了アドレス>－<スタートアドレス0100H>)
-N ALPHA.COM↵ ……………実行形式のプログラム名をセット(「.EXE」ではなく「.COM」にする)
-W↵ ……………プログラムの書き出し
Writing 005C bytes
-Q↵ ……………DEBUGを終了してMS-DOSに戻る

A>DIR ALPHA.COM↵ …………実行型ファイルが作成されているかどうかを確認する

 ドライブ A: のディスクにはボリュームラベルがありません
 ディレクトリは A:¥

ALPHA    COM     92  87-01-18   4:50 …………92バイトの小さな実行型
                                          ファイルが作成された
          1 個のファイルがあります
     76800 バイトが使用可能です

A>ALPHA↵ …………実行型ファイルが正しく動作するかどうか，実行して確めてみる
C  アルファベット大文字である
3  アルファベット大文字でない
d  アルファベット大文字でない
^C …………MS-DOSのコマンドモードに戻る

A>
```

図 実習10のプログラムをMS-DOSの実行型ファイルにする

このようにMS-DOSの実行型ファイルは，DEBUGを使ってこんなに簡単に作れるのです．なお，7章の「やさしいプログラミングの実例」でも，同様の方法でMS-DOSの実行型ファイルを作成しています．

LOOP 命令

条件ジャンプ命令の一種として，LOOP 命令という繰り返し処理専用の命令が用意されています．〝同じことを 10 回繰り返す〟などといった繰り返し処理は，プログラムを作る上でもよく出てくる処理です．これは，今までに出てきた命令を使うと次のように書くことができます（図 6-27）．

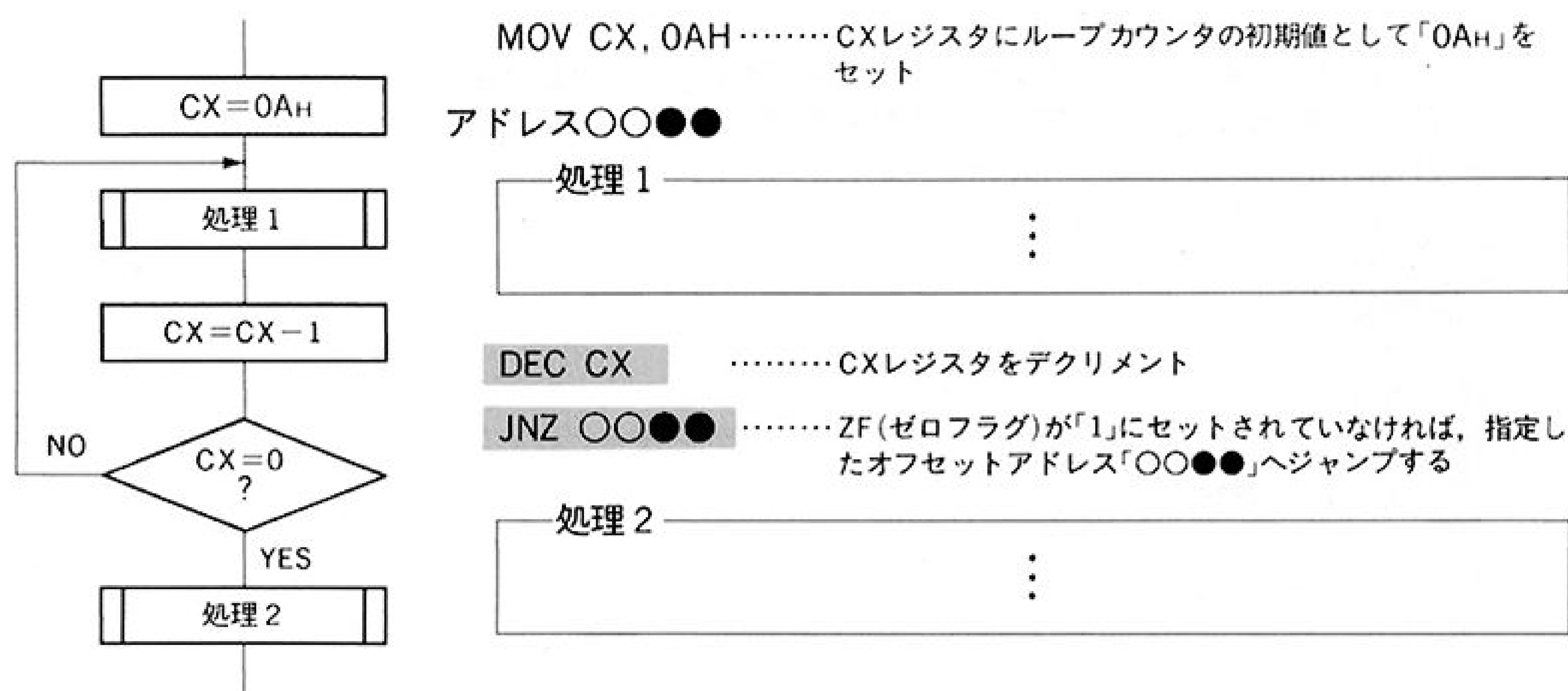


図 6-27　条件ジャンプ命令を使った繰り返し処理

つまり，カウンタとして使う CX レジスタに繰り返し回数をセットしておき，処理が 1 回終わるとその内容をデクリメントして，0 になるまで繰り返すというものです．これは典型的な繰り返し処理の書き方で実際に非常によく使われるので，8086CPU では CX レジスタを使った繰り返し処理専用の命令が用意されています．

それが LOOP 命令で，図 6-27 で色を付けてある 2 つの命令を 1 命令で実行します*．この命令は，

ループ(LOOP)
LOOP　○○●● …… **CX レジスタの内容をデクリメントし，0 でなければオフセットアドレス「○○●●」へジャンプする．**

という形式で表します．LOOP 命令は，まず CX レジスタの内容をデクリメントします．その結果が 0 でなければ指定されたオフセットアドレスにジャ

*ただしフラグの状態は変化しない．

ンプし，0であれば次の命令へと進みます．したがって，CXにセットされた回数だけ処理を繰り返すことになるのです．

CXレジスタは「カウントレジスタ」と呼ばれるように，繰り返しなどの回数をセットするカウンタとして使用されるレジスタです．6.6節で解説するストリング操作命令も，CXレジスタをカウンタとして用います．

実習11　LOOP命令による繰り返し

実習11のプログラムは，「オフセットアドレス0200_Hから始まるメモリに00_H，01_H，02_H…と順に増加する1バイトデータを128回繰り返して書き込む」というものを作ります．まず，DIレジスタをメモリへのポインタとして使用することにして，オフセットアドレス「0200_H」をロードします．CXレジスタはカウンタとして使用するので，繰り返し回数128（0080_H）をセットします．プログラムはALレジスタをインクリメントしながら，その値をDIレジスタの指すメモリにストアしていくという作業を繰り返します．

実習11のプログラムを次に示しましょう．

```
        MOV  AL , 0  ……  ALレジスタに1バイトデータ「0」をロードする．
        MOV  DI , 0200H …  DIレジスタに1ワードデータ「0200H」をロードする．
        MOV  CX , 0080H …  CXレジスタに1ワードデータ「0080H」をロードする．
  ┌──→ [ループの始め]
  │     MOV  [DI] , AL …  DIレジスタの内容をオフセットアドレスとするメモリ
  │                       にALレジスタの内容をストアする．
 128回  INC  AL ………………  ALレジスタの内容をインクリメントする．
 繰り返す
  │     INC  DI ………………  DIレジスタの内容をインクリメントする．
  └──── LOOP [ループの始め] …  CXレジスタをデクリメントし，「0」になるまで繰り
                          返す．
        INT  20H …………  プログラムを終了し，DEBUGへ戻る．
```

このプログラムをDEBUGで入力し，実行して結果を確かめてみたのが次の図6-28です．

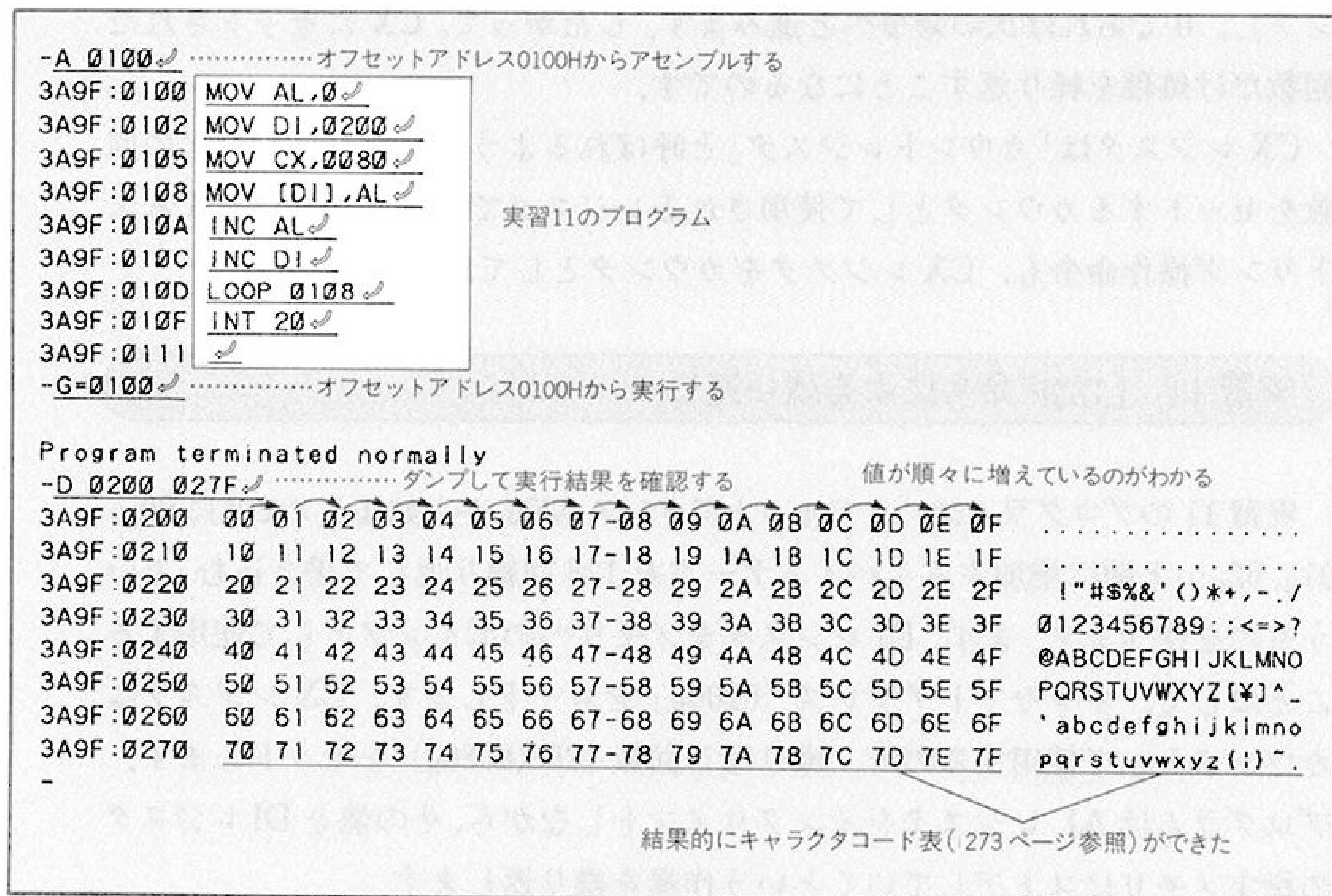


図 6-28　実習 11 の実行結果

ショートジャンプとニアジャンプ

ジャンプ命令の飛び先のアドレスは，マシン語では相対アドレスで表されます（164 ページのコラム参照）．相対アドレスは符号付きの数値として扱われますが(つまり，前にも後ろにも飛べるということ)，1バイトの場合と2バイトの場合があります．2バイトでは「−32768～＋32767」の数値を表すことができるので，結局すべてのオフセットアドレスへジャンプすることができます．ところが，1バイトでは「−128～＋127」の数値しか表すことができないので，ジャンプ命令の次の命令から数えてその範囲内にあるオフセットアドレスへしかジャンプすることができません．

相対アドレスを1バイトで表すジャンプをショート（SHORT）ジャンプ，2バイトで表すジャンプをニア（NEAR）ジャンプと呼びます．そして大事なことは「条件ジャンプ（LOOP 命令を含む）はすべてショートジャンプである」ということです．つまり，条件ジャンプを使うと，ジャンプ命令の次

の命令から－128～＋127バイト目の範囲内にあるオフセットアドレスにしかジャンプできないのです.

したがって条件ジャンプを使用する場合には，飛び先が相対アドレスの「－128～＋127」で表せる範囲内にあるかどうかに注意しなくてはなりません．もしも届かない場合には，届く範囲内の無条件ジャンプ命令を経由するなどの工夫が必要になります.

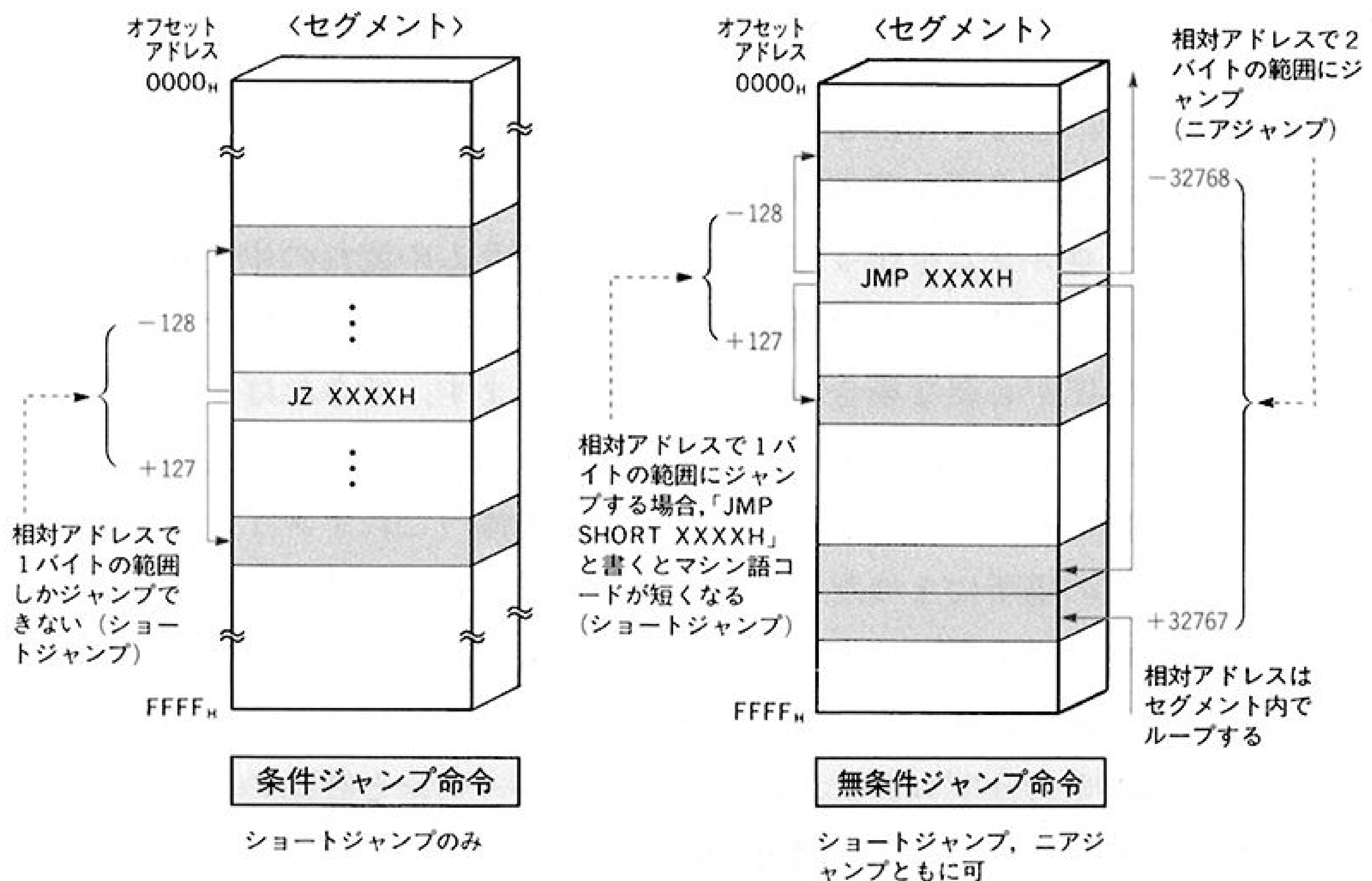


図6-29 ショートジャンプとニアジャンプ

無条件ジャンプではショートジャンプ，ニアジャンプのどちらを指定することもできます．また，本書では扱いませんがセグメントを超えるファー(FAR)ジャンプも可能です．アセンブリ言語では，特に指定がなければ無条件ジャンプ命令はニアジャンプであると仮定されていますが，ショートジャンプで届く範囲内にある場合には，

JMP　SHORT オフセットアドレス

としてショートジャンプを指定することもできます*.

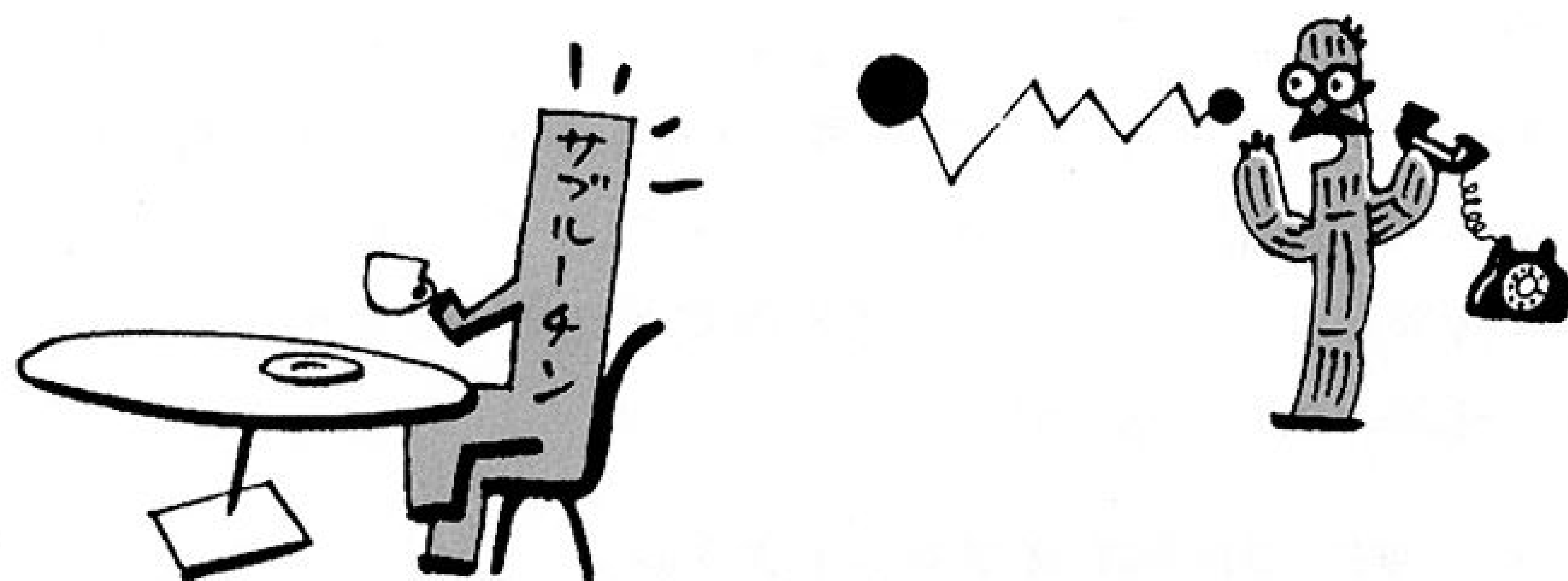
*積極的にショートジャンプを使うことによって，プログラムのバイト数を多少なりとも小さくすることができます．ただし実行速度には無関係です.

6.5 サブルーチンのコール/リターンとスタックのプッシュ/ポップ

サブルーチンは，プログラミング・テクニックのなかでもたいへん重要な概念です．プログラムを効率よく開発するためにも，読みやすいプログラム* を作成するためにもなくてはならない考え方であると言えるでしょう．

サブルーチンが必要な理由の１つは，「同じことを何度も書かずにすませる」ためです．プログラムを作っていると，プログラムの流れの中の異なる場面でまったく同じ処理をする必要があったり，ほとんど同じ処理だがちょっと違うという処理が必要な場合がしばしば出てきます．できればそういった処理は共用できるものを１つ書いておき，次からはそれを利用したいものです．

そこで，何度も出てくる処理はプログラム本体（これをメインルーチンと呼ぶ）とは別の場所に１つだけ作ることにして，その処理が必要になったときにそれを呼び出すという方法が考え出されました．このようにプログラム本体とは別の場所に作られ，プログラム本体から呼び出されるためのあるまとまった処理を行うプログラムのことをサブルーチンと呼ぶのです．このことをわかりやすく表したのが図 6-30 です．



*読みやすいプログラムとは，他人が見ても理解しやすいプログラムということを意味する．プログラム作成者自身も時間がたてばプログラムの内容を忘れてしまうので，読みやすさは重要である．

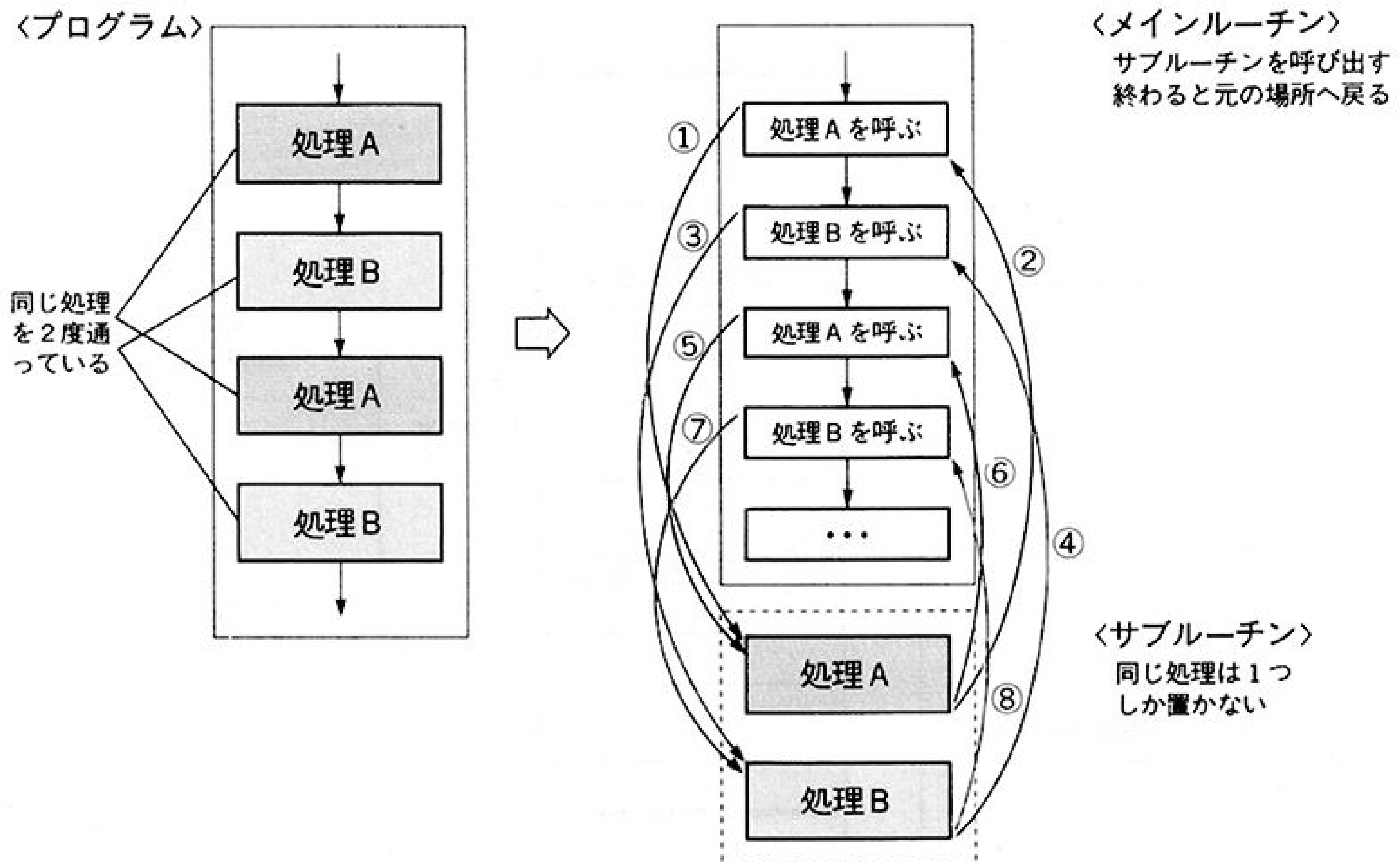


図6-30　サブルーチンの概念

サブルーチンが必要なもう１つの理由は,「プログラムの構造をわかりやすくする」ためです．たとえ一度しか実行しない処理であっても，ある程度まとまったものならばサブルーチンにしてプログラム本体の外に作るのが，プログラミングの常套手段とされています．１つのまとまった処理になると，通常数十個から数百個という数の命令からなります．それをプログラム本体から追い出してしまい,「これこれをする処理（サブルーチン）を呼び出す」という命令を１つだけプログラム本体に置くことによって，プログラム全体の見通しをよくするのです．１つの処理を呼び出して，次の処理を呼び出して，というぐあいに１つの処理単位が１つの命令で置き換えられるので，プログラム全体の流れが容易に理解できます．

このようにプログラムの構造をわかりやすくすることを,プログラムの「構造化」と呼んでいます．

CPUの命令では,メインルーチンからサブルーチンを呼び出すCALL（コール）命令と，サブルーチンからメインルーチンへ戻るRET（リターン）命令が用意されています．この２つの命令は必ず対になって使われます．この命令を次の図6-31に図解してみましょう．

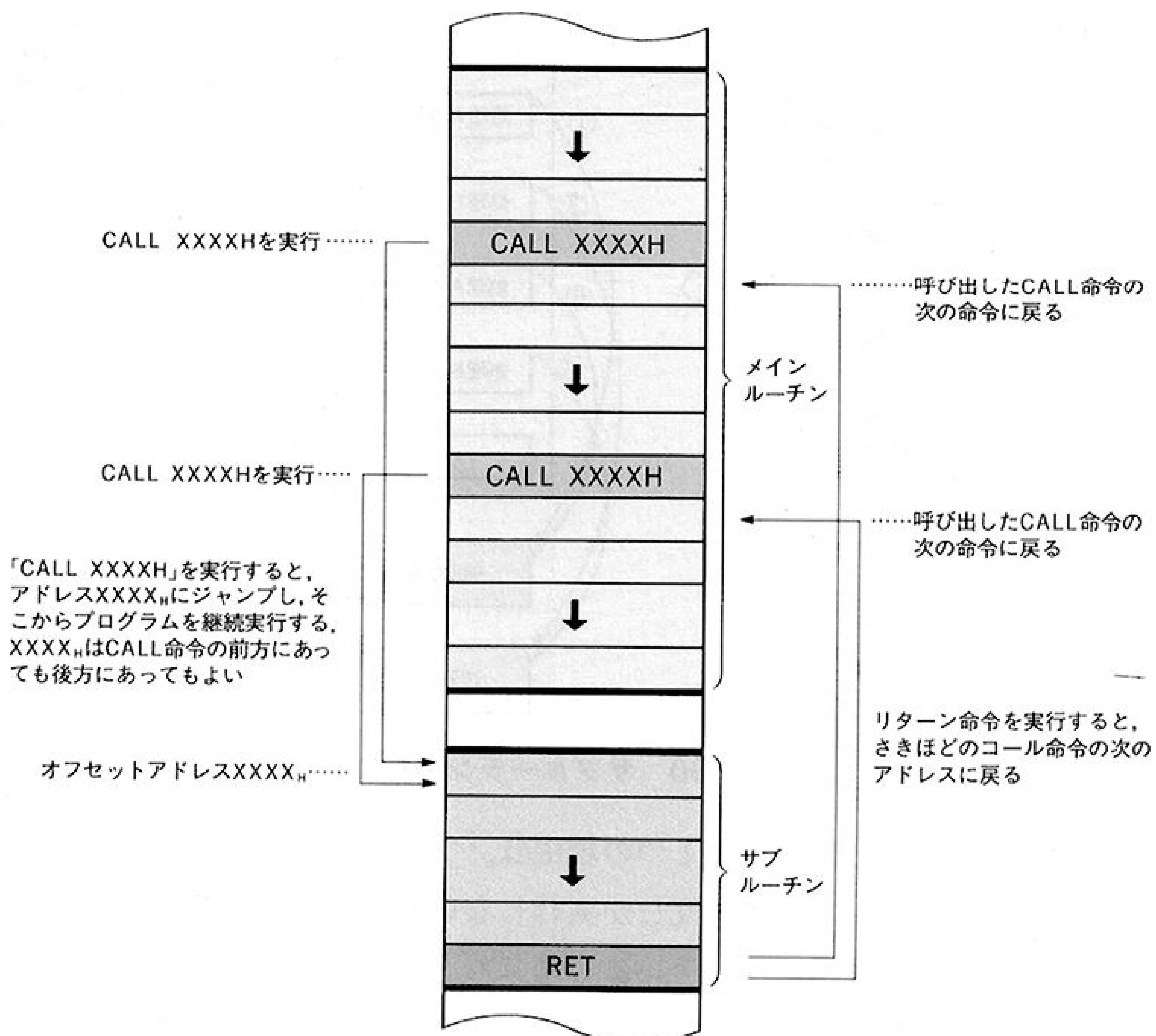


図 6-31　コール命令とリターン命令の図解

この図からもわかるように，コール命令は自動的にもとの位置に戻ってくることができるジャンプ命令と考えてもよいでしょう．もとの位置（正確にはコール命令の次の命令）に戻るための合図がリターン命令の実行です．

コール命令は，図のようにメインルーチンからサブルーチンをコールしますが，サブルーチンの中にまたコール命令があって，さらにサブルーチンをコールしてもかまいません．呼び出されるのはサブルーチンのサブルーチンになるわけです．さらに，そのまたサブルーチンというぐあいに，後述するスタックに余裕のある限り重ねて呼び出すことができます．サブルーチンからさらにサブルーチンを呼ぶことを「入れ子構造」とか，「ネスティングする」などと言います．サブルーチンのサブルーチンであっても，リターン命令では必ずそれを呼び出したところへ戻ります．このような「入れ子構造」を図で示してみましょう（図 6-32）．

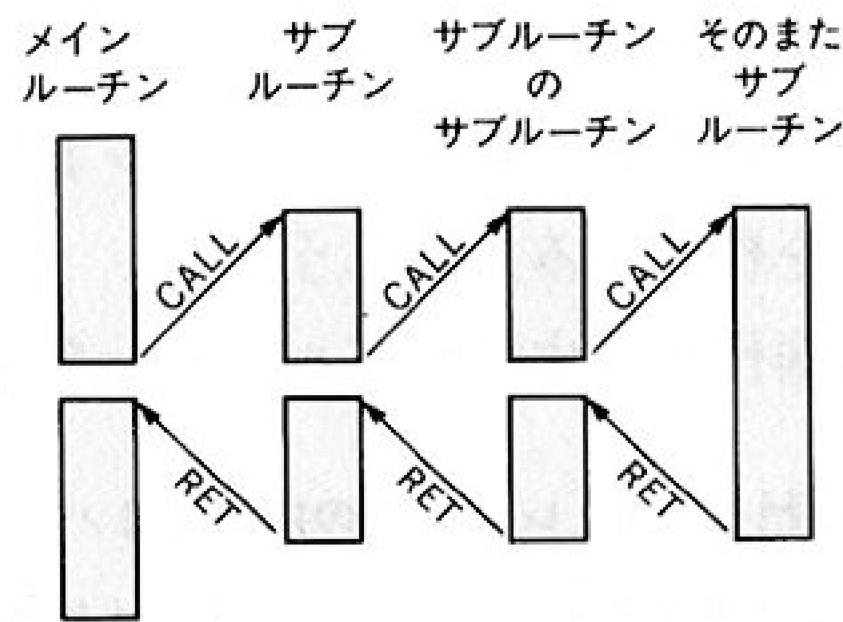


図 6-32　サブルーチンのネスティング（入れ子構造）

最後に，コール命令とリターン命令の書式をまとめておきます．

コール(CALL)
CALL　○○●●　……　オフセットアドレス「○○●●」から始まるサブルーチンを呼び出す．

リターン(RETurn)
RET　……　サブルーチンを呼び出したCALL命令の次の命令に戻る．

実習 12　サブルーチン・コール

「実習 10」のキーボードから入力した文字がアルファベット大文字かどうかを調べるプログラムを使って，コール命令とリターン命令の実習を行います．実習 10 の内容を忘れてしまった方はもう一度 171 ページを見てください．この実習では次のような判別作業を1つのサブルーチンとします．

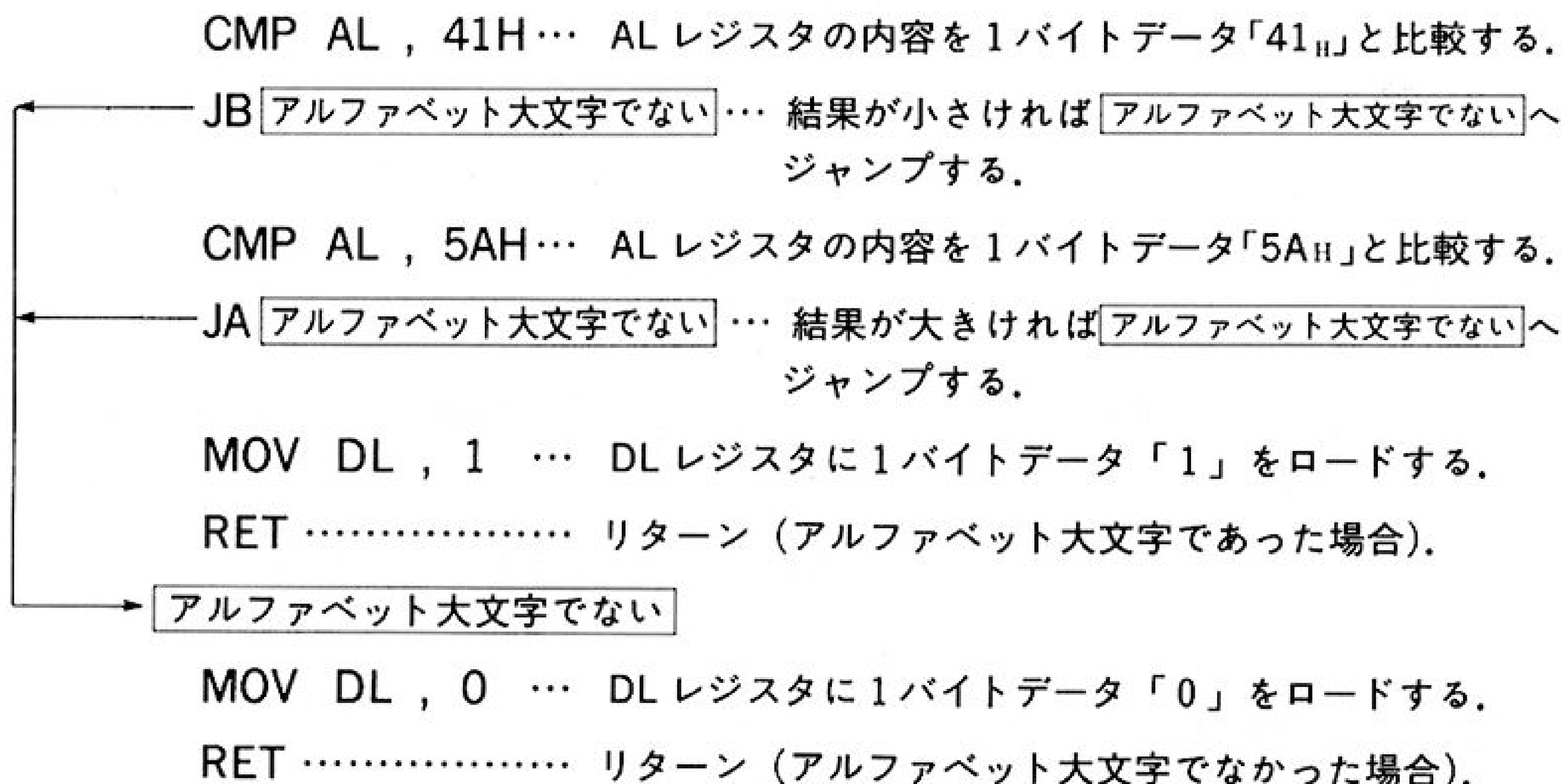
入力された文字がアルファベット大文字かどうかを調べる．すなわち入力された文字のキャラクタコード（AL レジスタの内容）を「A」および「Z」のキャラクタコードと比較して，その範囲内にあれば DL レジスタに 1 をセットし，範囲外にあるときには DL レジスタに 0 をセットする．

このサブルーチンでは，キーボードから入力された文字に限らず，メモリ上に格納されている文字列の文字がアルファベット大文字かどうかを調べるという使い方もできます．

メインルーチンでは，AL レジスタに適当な文字のキャラクタコードやキーボードから入力された文字のキャラクタコードをセットしてこの判別用サブ

ルーチンをコールします．サブルーチンから戻るとDLレジスタに判別の結果がはいっているので，それを適当なメモリにセットします．これを数回繰り返すようなプログラムを作ってみましょう．

まず最初に上記の判別作業をするサブルーチンを作成します．



このプログラムをオフセットアドレス「0200H」のメモリに置き，「0100H」からのメインルーチンで次のようにコールしましょう．

```
MOV AL , 58H ………… ALレジスタに「X」のキャラクタコード（58H）をロードする．
CALL 判別サブルーチン … 判別サブルーチンをコールする．
MOV [0300H], DL …… 結果をオフセットアドレス「0300H」のメモリにストアする．
MOV AL , 38H ………… ALレジスタに「8」のキャラクタコード（38H）をロードする．
CALL 判別サブルーチン …… 判別サブルーチンをコールする．
MOV [0301H], DL …… 結果をオフセットアドレス「0301H」のメモリにストアする．
MOV AH , 1 ┐
INT 21H    ┘ ファンクションコール1番(キーボードから1文字入力する)．
CALL 判別サブルーチン …… 判別サブルーチンをコールする．
MOV [0302H], DL …… 結果をオフセットアドレス「0302H」のメモリにストアする．
INT 20H ………………… プログラムを終了しDEBUGに戻る．
```

これでサブルーチンとメインルーチンができました．このプログラムをDEBUGで入力し，実行して結果を確かめたのが次の図 6-33 です．

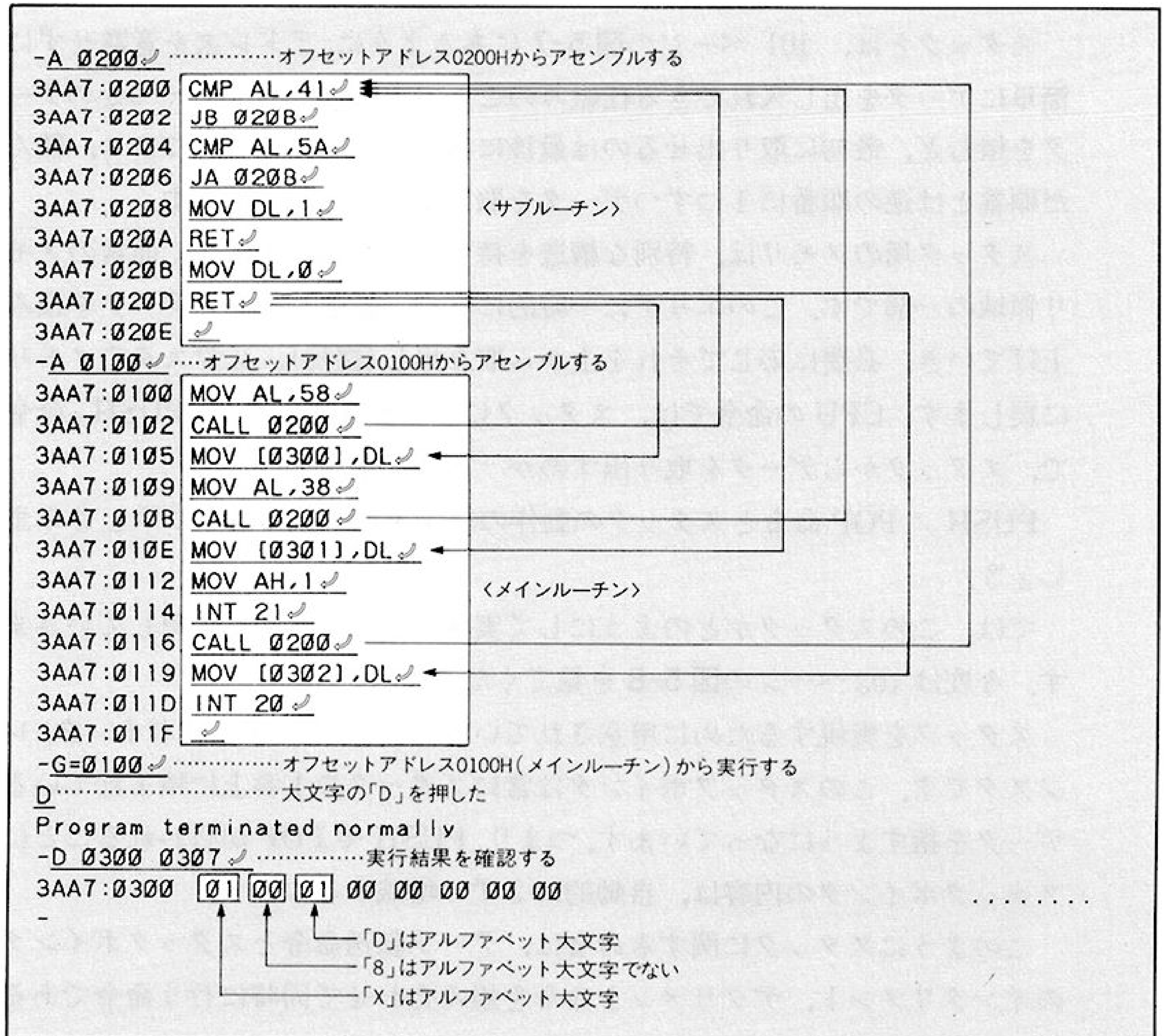


図 6-33　実習 12 の実行結果

スタックのプッシュ／ポップ

スタックの概念については「5.4 スタックとその働き」で解説してきましたが，ここでは少し具体的にお話することにしましょう．

スタックとは，101 ページの図 5-7 にあるように，アドレスを意識せずに簡単にデータを出し入れできる仕組みのことです．スタックにいくつかのデータを積むと，最初に取り出せるのは最後に積んだデータです．つまり，積んだ順番とは逆の順番に 1 つずつデータを取り出すことができます．

スタック用のメモリは，特別な構造を持っているわけではなく通常のメモリ領域の一部です．このエリアに一時的に保存（退避）したいデータを積み上げていき，必要に応じてそれを上から取り出し(復帰)，レジスタやメモリに戻します．CPU の命令では，スタックにデータを積むのが「PUSH（プッシュ）」命令で，スタックからデータを取り出すのが「POP（ポップ）」命令です．

PUSH ／ POP 命令とスタックの動作の様子を次の図 6-34 に示してみましょう．

では，このスタックがどのようにして実現されているかを説明していきます．今度は 103 ページの図 5-8 を見てください．

スタックを実現するために用意されているのが SP（スタックポインタ）レジスタです．このスタックポインタは常にスタックの 1 番上に積まれているデータを指すようになっています．つまり，PUSH や POP が行われるごとにスタックポインタの内容は，自動的に 2 ずつ増減するのです．

このようにスタックに関する命令は，データ転送命令とスタックポインタのインクリメント，デクリメント命令を組み合わせて同時に行う命令であるといえます．

PUSH ／ POP 命令の書式を AX レジスタを例にして，以下でまとめて示しておきます．

プッシュ (PUSH)
PUSH　AX …… スタックポインタを 2 減らし，その値をオフセットアドレスとするスタック領域に AX レジスタの内容を退避．

ポップ (POP)
POP　AX …… スタックポインタの値をオフセットアドレスとするスタック領域から AX レジスタに 1 ワードデータを復帰し，スタックポインタに 2 を加える．

また，フラグレジスタに関しては特別に，

プッシュフラグ(PUSH Flag)
PUSHF　……　スタックポインタを2減らし，その値をオフセットアドレスとするスタック領域にフラグレジスタの内容を退避．

ポップフラグ(POP Flag)
POPF　……　スタックポインタの値をオフセットアドレスとするスタック領域からフラグレジスタに1ワードデータを復帰し，スタックポインタに2を加える．

という形式で表します．

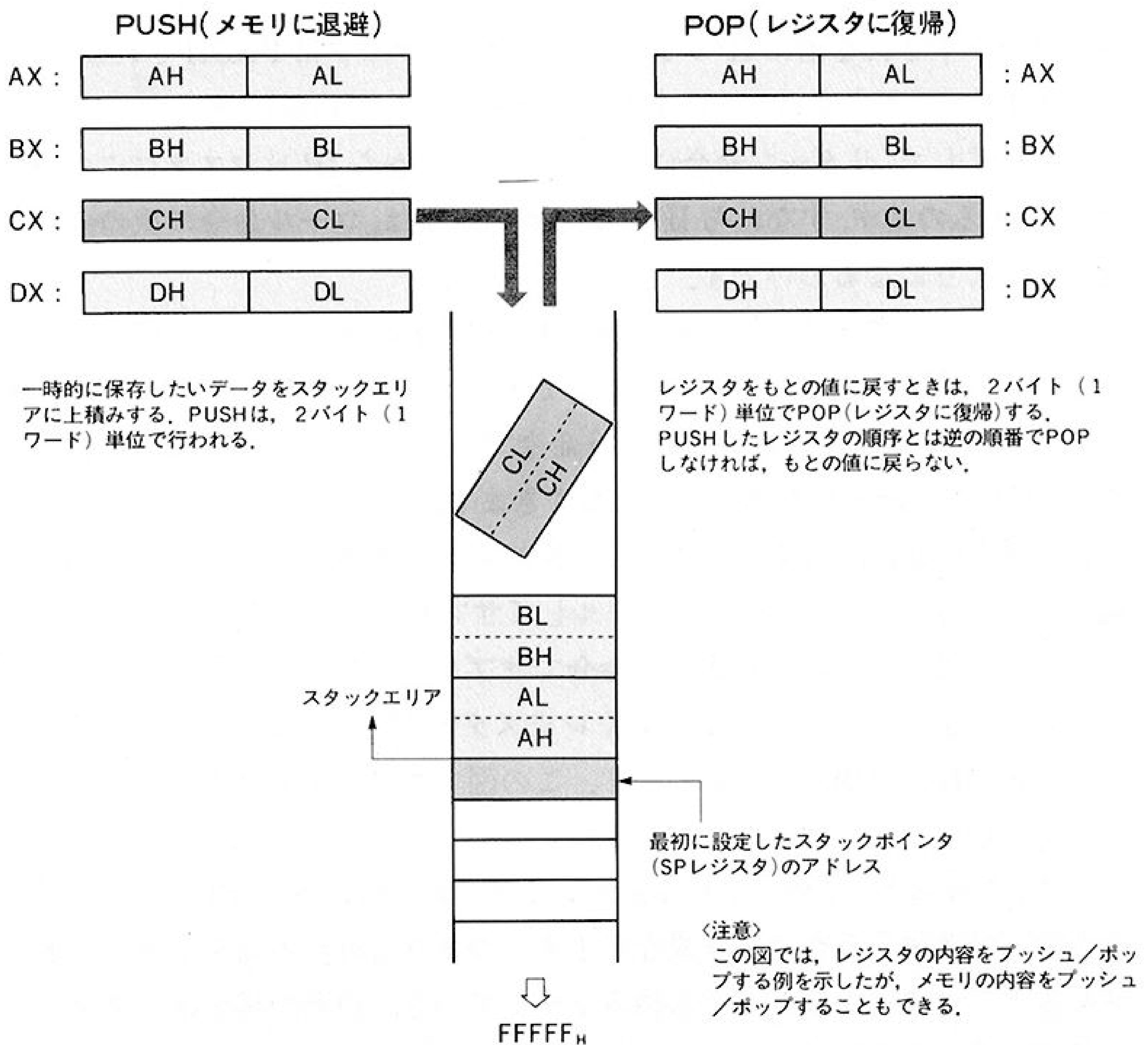


図6-34　PUSH／POP命令とスタックの動作

コール／リターン命令とスタック

コール命令とリターン命令を使うと，なぜ呼び出したところに復帰できるのでしょうか？　実は，コール命令とリターン命令は，前述のスタックを使って実現されているのです．

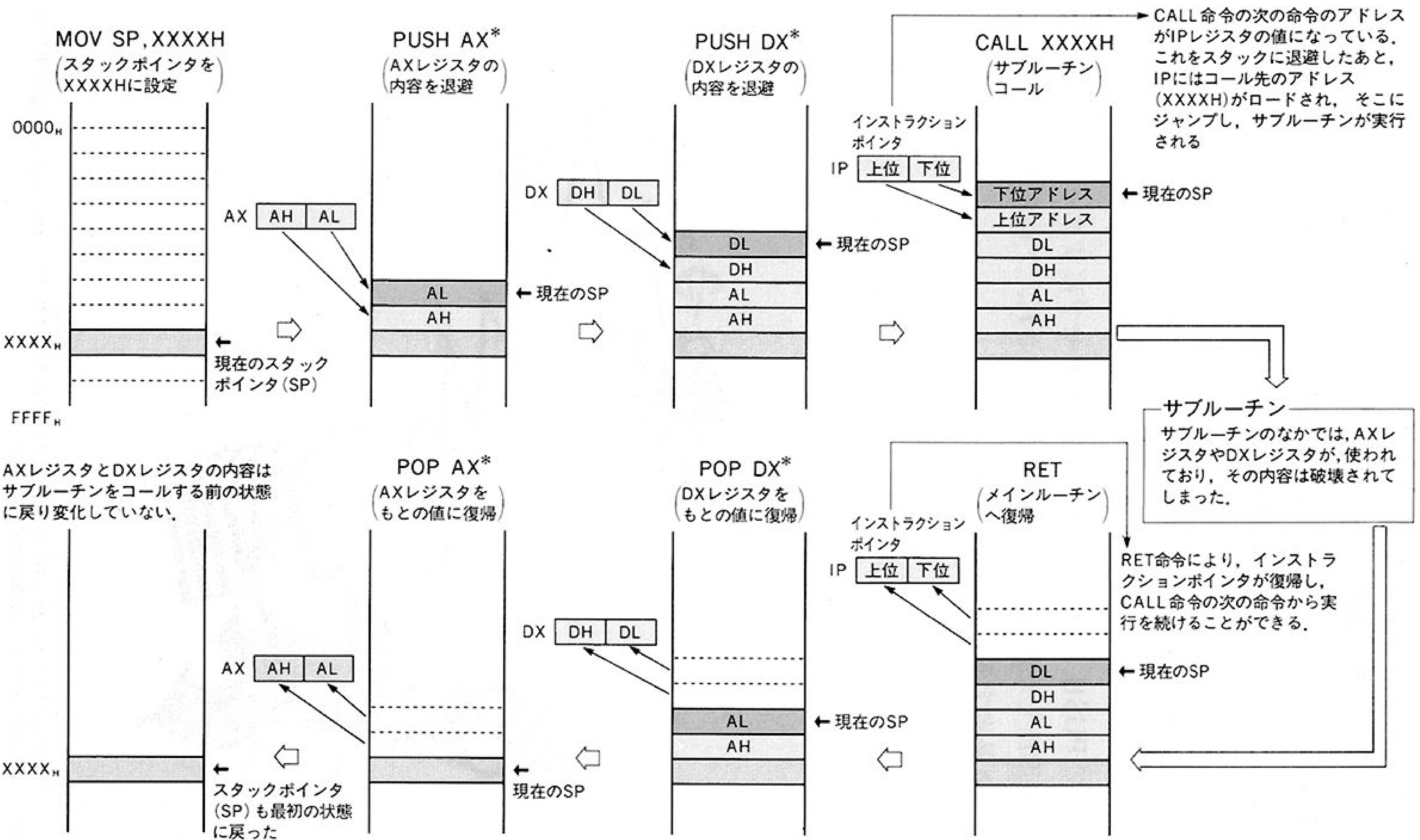
これまでも述べてきたように，プログラムの実行順序はIP（インストラクションポインタ）レジスタが管理しており，たとえばジャンプ命令を実行すると，IPレジスタにジャンプ先のオフセットアドレスの値が強制的にロードされて，そのアドレスからプログラムの実行が続けられます．

コール命令の場合は，IPレジスタにジャンプ先のオフセットアドレスが強制的にロードされる前にIPレジスタの内容がスタックにPUSHされます（このときIPレジスタはコール命令の次の命令を指している）．そしてサブルーチンが終了して，リターン命令にくるとスタックからIPレジスタにこの値がPOPされるのです．すなわちIPレジスタの内容は，コール命令の次の命令を再び指すようになるわけです．

このようにしてメインルーチンから呼び出されたサブルーチンは，もとのメインルーチンへ戻ることができるのです．コール命令はジャンプ命令とIPレジスタのPUSH命令を同時に行う命令であり，リターン命令はIPレジスタをPOPする命令だと考えることもできます．

次の図6-35は，「AXレジスタとDXレジスタの内容をスタックエリアに退避しておき，サブルーチンをコールしてサブルーチンの中でAXやDXのレジスタを使い，その後リターン命令でサブルーチンからメインルーチンに戻る際に，退避していたAXやDXレジスタの内容をもとに復帰させる」という一連の操作を図解したものです．この図でスタックエリアとスタックポインタの役割をよく理解してください．

スタック領域として何バイト必要かは，そのプログラムの中でPUSHやCALL命令が何回連続するかにより異なります．つまりこれらの命令が次々と実行されると，スタックはどんどん積み上がっていき，最悪の場合は，プログラム領域やデータ領域にまでくい込んでその内容を破壊してしまうことになります．



＊この図では理解しやすいように，PUSH／POP命令をサブルーチンを呼び出す前後で実行しているが，本来はサブルーチンで使用するレジスタはサブルーチンの中でPUSH(退避)，POP(復帰)を行うのが普通である。

図6-35　スタックに関する各種の命令を実行した場合のスタックの動作

通常のプログラムでは，その先頭でスタック領域を設定する，つまりスタックポインタにスタック領域最後のオフセットアドレスを設定します(MOV SP, XXXXH 命令で行う)．しかし，本書の実習のようにDEBUG上でプログラムを作成する場合は，すでにDEBUGによってスタックエリアが確保されているので，わざわざ設定する必要はありません*．

また，スタックにPUSHしたりスタックからPOPしたりする場合に，その対応が正しくとれるようにするのはプログラムを作る人の責任です．サブルーチン内でPUSHしたにもかかわらずPOPせずにリターンしようとすると，先にPUSHした値がIPレジスタにロードされ，プログラムは意図しないところへと暴走することになります．



* 174ページのコラムのようにMS-DOSの実行型ファイルとして作成し，MS-DOSから直接実行する場合には，MS-DOSがスタック領域の設定を行う．

6.6 ストリング操作命令

ストリング(string)とは，データの列のことを意味します．連続したメモリに対してよく行われる操作を，高速に実行できるように特に用意されているのがストリング操作命令です．この機能は8086CPUの大きな特徴であり，その能力を高めているものだと言ってもよいでしょう．

実習 13　ブロック転送

大量のデータをあるメモリ領域から別のメモリ領域へ転送することが，実際のプログラミングではよく行われます．連続したメモリの内容を，別の連続したメモリ領域に転送することを「ブロック転送」と呼びます．

```
MOV SI , 0000H … 転送元のオフセットアドレスをSIレジスタにロードする.
MOV DI , 1000H … 転送先のオフセットアドレスをDIレジスタにロードする.
MOV CX , 0100H… 転送するバイト数をCXレジスタにロードする.
┌─→[繰り返し]
│   MOV AL ,[SI]    …… SIレジスタの内容をオフセットアドレスとするメモリの内容をALレジスタにロードする.
256回 MOV [DI], AL  …… ALレジスタの内容をDIレジスタの内容をオフセットアドレスとするメモリにストアする.
繰り返す
│   INC SI ……………… SIレジスタの内容を＋1する.
│   INC DI ……………… DIレジスタの内容を＋1する.
└── LOOP [繰り返し] ……… CXレジスタの内容を－1し，その値が0でなければ[繰り返し]にジャンプする.
INT 20H …………… プログラムを終了し，DEBUGに戻る.
```

ここでは，ブロック転送のために用意されているストリング命令を実習します．まず，このブロック転送を行うプログラムを今までに実習した命令を組み合わせて作ってみましょう．前ページで示したプログラムがどのような動作をするか考えてみてください．

このプログラムは，オフセットアドレス「0000_H」から始まる「0100_H」バイト（256バイト）のメモリの内容をオフセットアドレス「1000_H」から始まる「0100_H」バイトのメモリへ転送します．

このようなデータ転送（ブロック転送）はしばしば必要になるうえ，CPUの処理時間が多くかかるので，8086CPUでは専用の命令が用意されています．先のプログラムで色を付けた4つの命令は，次の1命令で置き換えることができます．

ムーブストリングバイト (MOVe String Byte)
MOVSB …… SIレジスタの指す1バイトのメモリの内容をDIレジスタの指すメモリに転送し，SIおよびDIレジスタをインクリメントまたはデクリメントする．

この命令はALレジスタを介さずに，SIレジスタの指すメモリからDIレジスタの指すメモリへと直接1バイトのデータを転送し，さらにSI, DIレジスタの内容をそれぞれインクリメントまたはデクリメントします．ですからALレジスタの内容は変化しません．この命令を使うと1バイト単位のブロック転送が，先の4命令の組合せよりも高速に実行されます．

さらに，この命令をCXレジスタにセットされた回数だけ繰り返すように指定することができます．そのためにはMOVSB命令の前にリピートプリフィックス*「REP」を付けます．

リピート (REPeat)
REP MOVSB …… MOVSB命令を実行後CXレジスタをデクリメントし，0になるまで繰り返す．

このリピートプリフィックスを付けると，MOVSB命令を実行してはCXレジスタの内容をデクリメントします．そしてそれをCXレジスタの内容が0になるまで繰り返します．

---

*プリフィックス（Prefix）とは接頭辞，つまり前に付けることばという意味がある．

MOVSB 命令が実行されるときの動作を次の図 6-36 に図示してみましょう．

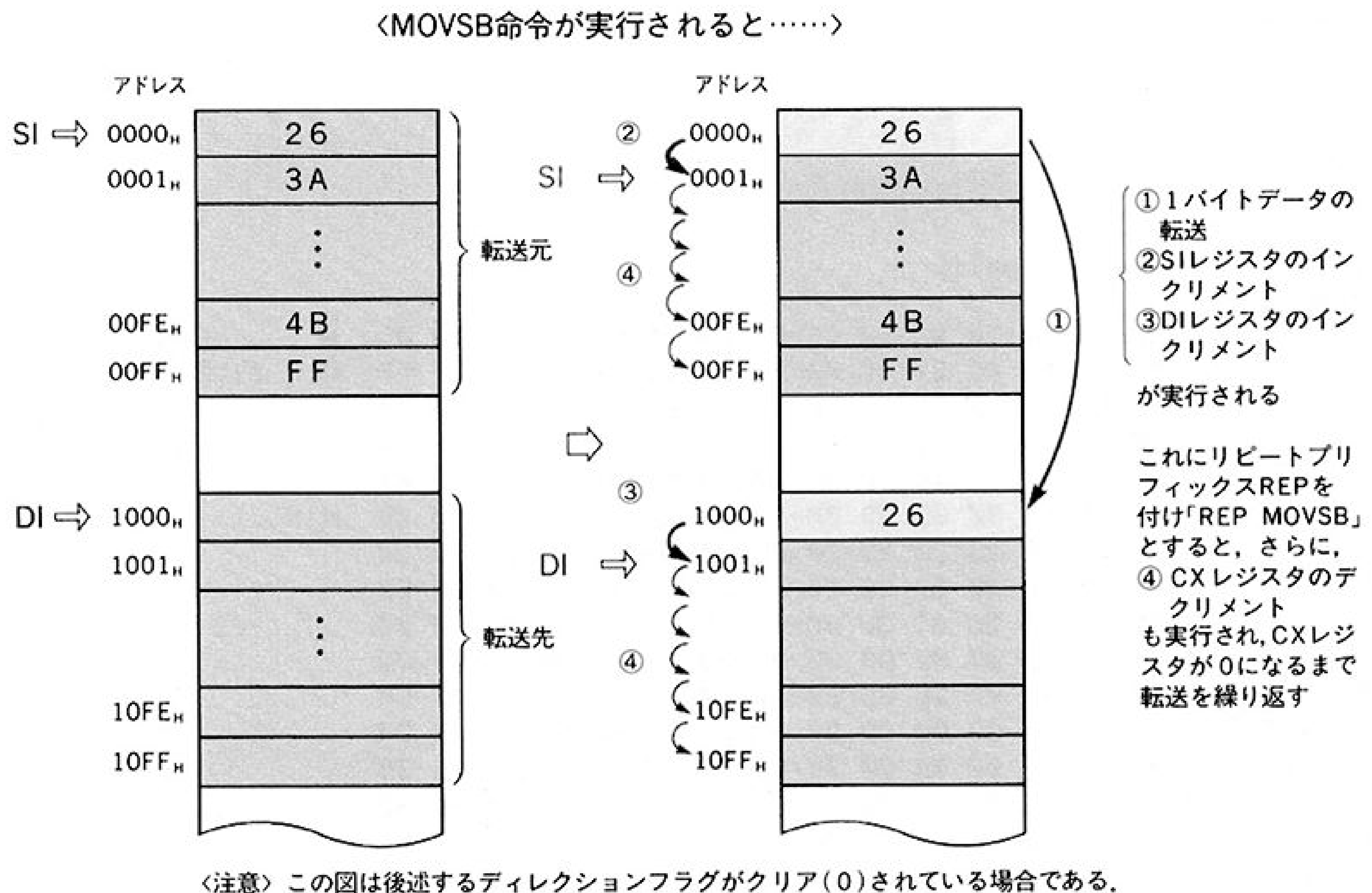


図 6-36　MOVSB 命令の動作

この命令を使って先のプログラムを作りなおすと，次のようになります．

```
MOV  SI , 0000H ･･･ 転送元のオフセットアドレスを SI レジスタにロードする．
MOV  DI , 1000H ･･･ 転送先のオフセットアドレスを DI レジスタにロードする．
MOV  CX , 0100H ･･･ 転送するバイト数を CX レジスタにロードする．
CLD ･･････････････････････ ディレクションフラグをクリアする．
REP  MOVSB ･････････ ブロック転送を実行する．
INT  20H･･････････････ プログラムを終了し，DEBUG へ戻る．
```

プログラムに「CLD」という命令を加えましたが，これについては後で解説するとして，このプログラムを DEBUG で入力し，実行して結果を確かめてみましょう．

```
-A Ø1ØØ↵ ……………オフセットアドレス0100Hからアセンブルする
3A9F:Ø1ØØ  MOV SI,ØØØØ↵
3A9F:Ø1Ø3  MOV DI,1ØØØ↵
3A9F:Ø1Ø6  MOV CX,Ø1ØØ↵
3A9F:Ø1Ø9  CLD↵                    実習13のプログラム
3A9F:Ø1ØA  REP MOVSB↵
3A9F:Ø1ØC  INT 2Ø↵
3A9F:Ø1ØE  ↵
-G=Ø1ØØ↵ ……………オフセットアドレス0100Hから実行する

Program terminated normally
-D ØØØØ ØØFF↵ ……………オフセットアドレス0000H～00FFHのメモリをダンプする
3A9F:ØØØØ  CD 2Ø ØØ AØ ØØ 9A 5Ø C3-D7 F3 28 Ø9 C2 31 C5 Ø9   M .  ..PCWs(.BIE.
3A9F:ØØ1Ø  C2 31 FØ Ø8 C2 31 CC 3Ø-Ø3 Ø4 Ø1 ØØ Ø2 FF FF FF   B1p.B1LØ........
3A9F:ØØ2Ø  FF FF FF FF FF FF FF FF-FF FF FF FF AD 31 15 Ø7   .............-1..
3A9F:ØØ3Ø  CC 3Ø 14 ØØ 18 ØØ 9F 3A-FF FF FF FF ØØ ØØ ØØ ØØ   LØ.....:........
3A9F:ØØ4Ø  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   ................
3A9F:ØØ5Ø  CD 21 CB ØØ ØØ ØØ ØØ ØØ-ØØ ØØ ØØ ØØ ØØ 2Ø 2Ø 2Ø   M!K..........
3A9F:ØØ6Ø  2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø-ØØ ØØ ØØ ØØ ØØ 2Ø 2Ø 2Ø           .....
3A9F:ØØ7Ø  2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ           ........
3A9F:ØØ8Ø  ØØ ØD 2Ø ØD 2D 32 32 ØD-ØD 36 2D 31 39 ØD ØØ ØØ   .. .-22..6-19...
3A9F:ØØ9Ø  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   ................
3A9F:ØØAØ  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   ................
3A9F:ØØBØ  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   ................
3A9F:ØØCØ  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   ................
3A9F:ØØDØ  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   ................
3A9F:ØØEØ  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   ................
3A9F:ØØFØ  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   ................
-D 1ØØØ 1ØFF↵ …オフセットアドレス1000H～10FFHのメモリをダンプする．そっくり転送されていることがわかる
3A9F:1ØØØ  CD 2Ø ØØ AØ ØØ 9A 5Ø C3-D7 F3 28 Ø9 C2 31 C5 Ø9   M .  ..PCWs(.BIE.
3A9F:1Ø1Ø  C2 31 FØ Ø8 C2 31 CC 3Ø-Ø3 Ø4 Ø1 ØØ Ø2 FF FF FF   B1p.B1LØ........
3A9F:1Ø2Ø  FF FF FF FF FF FF FF FF-FF FF FF FF AD 31 15 Ø7   .............-1..
3A9F:1Ø3Ø  CC 3Ø 14 ØØ 18 ØØ 9F 3A-FF FF FF FF ØØ ØØ ØØ ØØ   LØ.....:........
3A9F:1Ø4Ø  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   ................
3A9F:1Ø5Ø  CD 21 CB ØØ ØØ ØØ ØØ ØØ-ØØ ØØ ØØ ØØ ØØ 2Ø 2Ø 2Ø   M!K..........
3A9F:1Ø6Ø  2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø-ØØ ØØ ØØ ØØ ØØ 2Ø 2Ø 2Ø           .....
3A9F:1Ø7Ø  2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ           ........
3A9F:1Ø8Ø  ØØ ØD 2Ø ØD 2D 32 32 ØD-ØD 36 2D 31 39 ØD ØØ ØØ   .. .-22..6-19...
3A9F:1Ø9Ø  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   ................
3A9F:1ØAØ  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   ................
3A9F:1ØBØ  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   ................
3A9F:1ØCØ  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   ................
3A9F:1ØDØ  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   ................
3A9F:1ØEØ  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   ................
3A9F:1ØFØ  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   ................
-
```

図 6-37　実習 13 の実行結果

このプログラム中の，

CLD（クリアディレクションフラグ (CLear Direction flag)）……ディレクションフラグをクリアする．

という命令は，その名のとおりフラグレジスタ中のディレクションフラグ(DF)をクリア（リセット）する，つまり0にする命令です．これはブロック転送の方向（ディレクション）を決めるフラグです．このフラグが0であるか1であるかによってMOVSB命令が実行されるときに，ポインタとなるレジスタがインクリメントされるかデクリメントされるかが決定されるのです．

これとは逆にディレクションフラグを1にセットするのが，

セットディレクションフラグ (SeT Direction flag)
STD　……　ディレクションフラグをセットする．

という命令です．

ディレクションフラグが，

0ならば（CLDの状態），ポインタをインクリメント
1ならば（STDの状態），ポインタをデクリメント

となります．なぜデクリメントする必要があるのかというと，ブロック転送するメモリ領域の転送元ブロックと転送先ブロックが重なっている場合が考えられるからです．重なっていなければ，単にインクリメントしながら転送するだけでかまいません．しかし，2つのメモリブロックが重なっている場合には方向を考えなければなりません．方向によっては，転送する前に転送元ブロックに書き込みが行われてもとのデータが破壊されてしまいます．このことを表したのが次のページの図6-38です．

MOVSBは1バイト単位の転送命令でしたが，1ワード単位で転送を行う命令もあります．この命令は，

ムーブストリングワード
MOVSW　……　SIレジスタの指す1ワードのメモリの内容をDIレジスタの指すメモリに転送し，SIおよびDIレジスタを2ずつインクリメントまたはデクリメントする．

という形式で表します．使用するレジスタなどはMOVSB命令と同じで，1回の実行で1ワードのデータを転送し，SIおよびDIレジスタは2ずつインクリメントまたはデクリメントされます．したがってリピートプリフィックス(REP)を付けて繰り返し実行すると，CXレジスタの値が同じでもMOVSB命令に比べて倍の量を転送することになります．MOVSB命令とMOVSW命

令のクロック数(実行時間)は同じなので，MOVSW 命令を使えば大量のデータをより高速に転送することが可能です．ただし MOVSW 命令を使うときには，転送するデータが偶数個でなければならないことに注意してください．

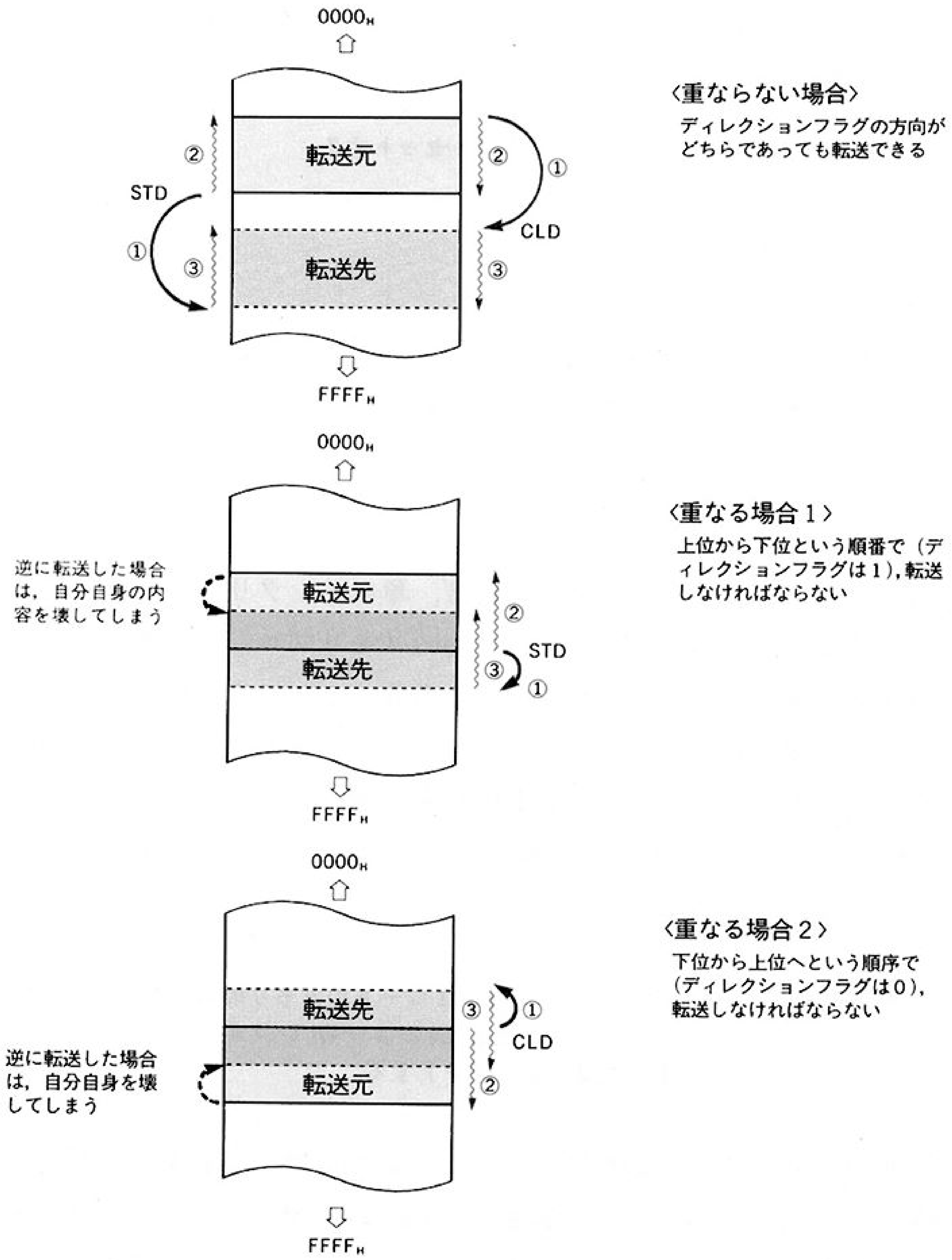


図 6-38　MOVSB 命令とディレクションフラグ

ストリング命令は，ほかにもいくつか用意されておりリピートプリフィックスと組み合わせたりすることによって，大量のデータを効率よく処理することができます．その他のストリング命令については，**表6-6**にストリング命令およびリピートプリフィックスの一覧とその機能を挙げておきます*．

〈ストリング操作命令〉

命　令	ニーモニック	動　作	機　　能	用途
(MOVe String) MOVS	MOVSB MOVSW	MOV [DI],[SI] INC(DEC) SI INC(DEC) DI	REPプリフィックスと組み合せて，SIで指定するブロック(転送元)から，DIで指定するブロック(転送先)へデータを転送する	ブロック転送
(LOaD String) LODS	LODSB LODSW	MOV AL,[SI] INC(DEC) SI	SIで指定するメモリからALレジスタにデータをロードする．REPプリフィックスと組み合せることはあまりない	データのロード
(STOre String) STOS	STOSB STOSW	MOV [DI],AL INC(DEC) DI	DIで指定するメモリにALレジスタの内容をストアする．また，REPプリフィックスと組み合せて，DIで指定するブロックへ，ALレジスタの内容を次々とストアする	データのストア ブロックのクリア
(CoMPare String) CMPS	CMPSB CMPSW	CMP [SI],[DI] INC(DEC) SI INC(DEC) DI	REPZプリフィックスと組み合せて，SIで指定するブロックとDIで指定するブロックを比較し，差異が見つかったところで停止する	ブロックの比較
(SCAn String) SCAS	SCASB SCASW	CMP [AL],DI INC(DEC) DI	REPNZプリフィックスと組み合せて，DIで指定するブロックとALレジスタの内容を比較し，同じデータが見つかったところで停止する	ブロック内のデータの検索

〈注意〉
- 各命令とも，転送するデータをバイトとして扱うかワードとして扱うのかによって，使用するニーモニックを使いわける．最後に「B」の付いたものがバイト単位，「W」の付いたものがワード単位となる．なお，ワード単位の場合，レジスタはALではなくAXが使用され，インクリメント，デクリメントは2ずつになる．
- SI／DIレジスタの増減の方向は，ディレクションフラグによって決まる．

〈リピートプリフィックス〉

ニーモニック	機　　能	組み合せて使うストリング命令
(REPeat) REP	ストリング操作命令を1回実行するたびに，CXレジスタをデクリメントし，0になるまで繰り返す	MOVS LODS STOS
(REPeat while Zero) REPZ	ストリング操作命令を1回実行するたびに，CXレジスタをデクリメントし，CXレジスタが0でなくゼロフラグが1である間，繰り返す	CMPS
(REPeat while Not Zero) REPNZ	ストリング操作命令を1回実行するたびに，CXレジスタをデクリメントし，CXレジスタが0でなくゼロフラグが0である間，繰り返す	SCAS

表6-6　ストリング命令とリピートプリフィックスの一覧表

＊ 5.6章で取り上げたサンプルプログラムは，ストリング命令を使うと実に簡単に書ける．各自で確めてみること

6.7 論理演算命令

6.3 節で算術演算命令について解説しましたが，CPU の行う演算にはそれ以外に論理演算命令と呼ばれる演算があります．論理演算とはその名のとおり論理的（ロジカル）に演算を行うものです．「1」か「0」，つまり「ある（1）」か「ない（0）」かが演算の対象になるのです．

よく使われる演算は，論理和（OR（オア））と論理積（AND（アンド））と呼ばれるものです．算術的な和や積との違いを次の**表 6-7** に示します．この表は，1 ビット単位での組合せによる算術および論理和／論理積の結果になっています．

算術和（+）	OR 論理和（∨）	算術積（×）	AND 論理積（∧）
0 + 0 = 0	0 ∨ 0 = 0	0 × 0 = 0	0 ∧ 0 = 0
0 + 1 = 1	0 ∨ 1 = 1	0 × 1 = 0	0 ∧ 1 = 0
1 + 0 = 1	1 ∨ 0 = 1	1 × 0 = 0	1 ∧ 0 = 0
1 + 1 = 2	1 ∨ 1 = 1	1 × 1 = 1	1 ∧ 1 = 1

→「ある（1）」と「ある（1）」の和は「ある（1）」

表 6-7　論理積（AND）と論理和（OR）

また，しばしば使われるものに排他的論理和（XOR（エクスクルーシブオア (eXclusive OR)））という論理演算があります．参考までにこの表も示しておきます（**表 6-8**）．

XOR 排他的論理和（∀）
0 ∀ 0 = 0
0 ∀ 1 = 1
1 ∀ 0 = 1
1 ∀ 1 = 0

表 6-8　排他的論理和（XOR）

論理和や論理積のマシン語命令は，それぞれ OR 命令，AND 命令となります．論理演算命令は，１バイトあるいは１ワード単位でその８ビットまたは 16 ビットのそれぞれのビットについて演算します．８ビットの場合について具体的な例を次に示しましょう（図 6-39）．

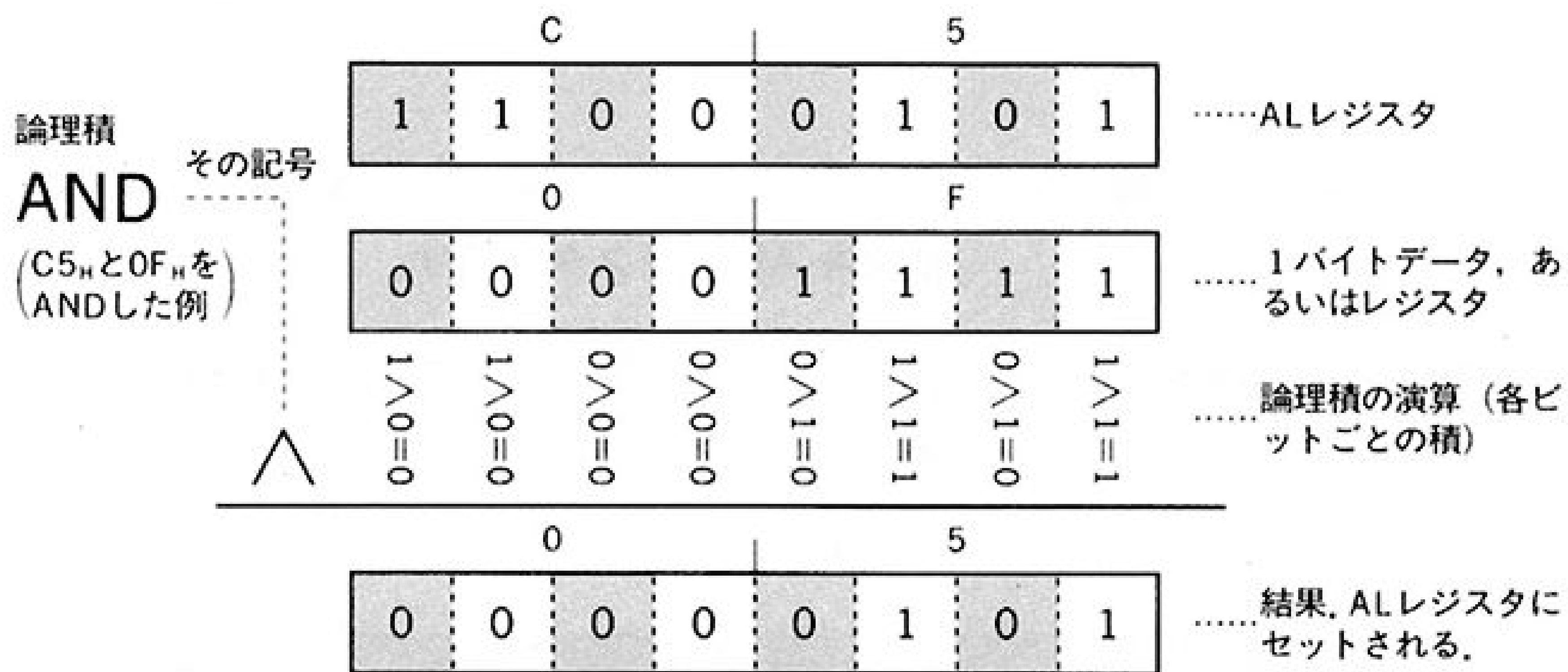


図 6-39　AND 命令の図解

このように AND 命令では，ある１バイトデータと「$0F_H$」の AND をとると，上位４ビットはすべて０になり，下位の４ビットはもとのまま残ることになります．このようなとき「上位４ビットをマスクした」と表現します．

OR 命令の実例も示してみましょう（図 3-40）．

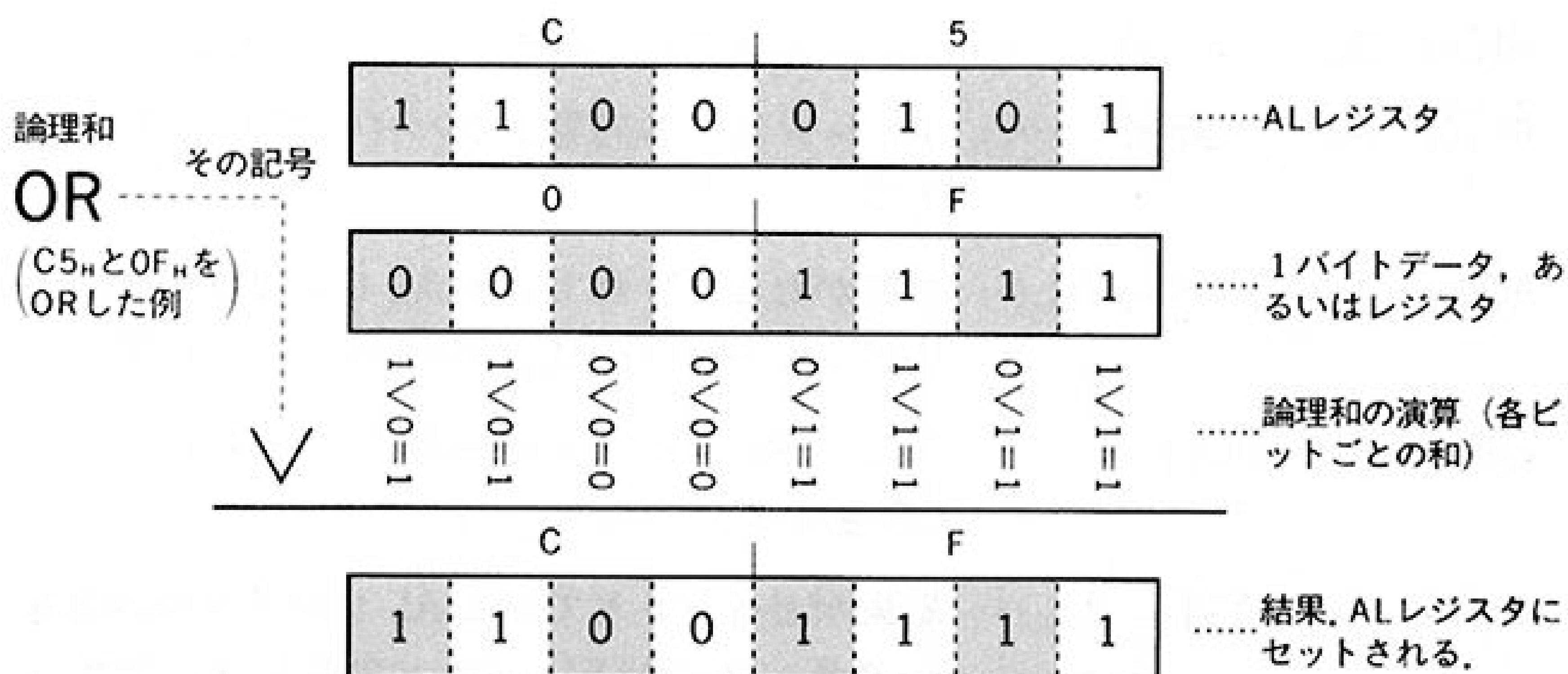


図 6-40　OR 命令の図解

このようにOR命令では，ある1バイトデータと「$0F_H$」のORをとると，上位4ビットはもとのままで，下位4ビットはすべて1になります．つまり「下位4ビットを1でマスクした」ことになります．

AND命令，OR命令の書き方は算術演算命令の書き方と同じです．たとえば，ALレジスタと1バイトデータのANDとORをとる場合の書式を次に示します．

AND（アンド） AL，●● …… ALレジスタと1バイトデータ「●●」のANDをとり，結果をALレジスタに残す．

OR（オア） AL，●● …… ALレジスタと1バイトデータ「●●」のORをとり，結果をALレジスタに残す．

実習14　8ビットデータの論理演算

実習14は次のようなプログラムを作成します．ALレジスタの値を「61_H」にセットし，「$5F_H$」とANDをとり，その結果をオフセットアドレス「0150_H」のメモリにストアします．さらにその値と「20_H」のORをとり，その結果をオフセットアドレス「0151_H」のメモリにストアします．

このプログラムは，次のようになります．

MOV AL，61H …… ALレジスタに1バイトデータ「61_H」をロードする．

AND AL，5FH …… ALレジスタの内容と1バイトデータ「$5F_H$」のANDをとる．

MOV [0150H]，AL… 結果がセットされているALレジスタの内容をオフセットアドレス「0150_H」のメモリにストアする．

OR AL，20H ……… さらにALレジスタの内容と1バイトデータ「20_H」のORをとる．

MOV [0151H]，AL… 結果がセットされているALレジスタの内容をオフセットアドレス「0151_H」のメモリにストアする．

これをDEBUGで入力し，実行した結果を次の図6-41に示します．

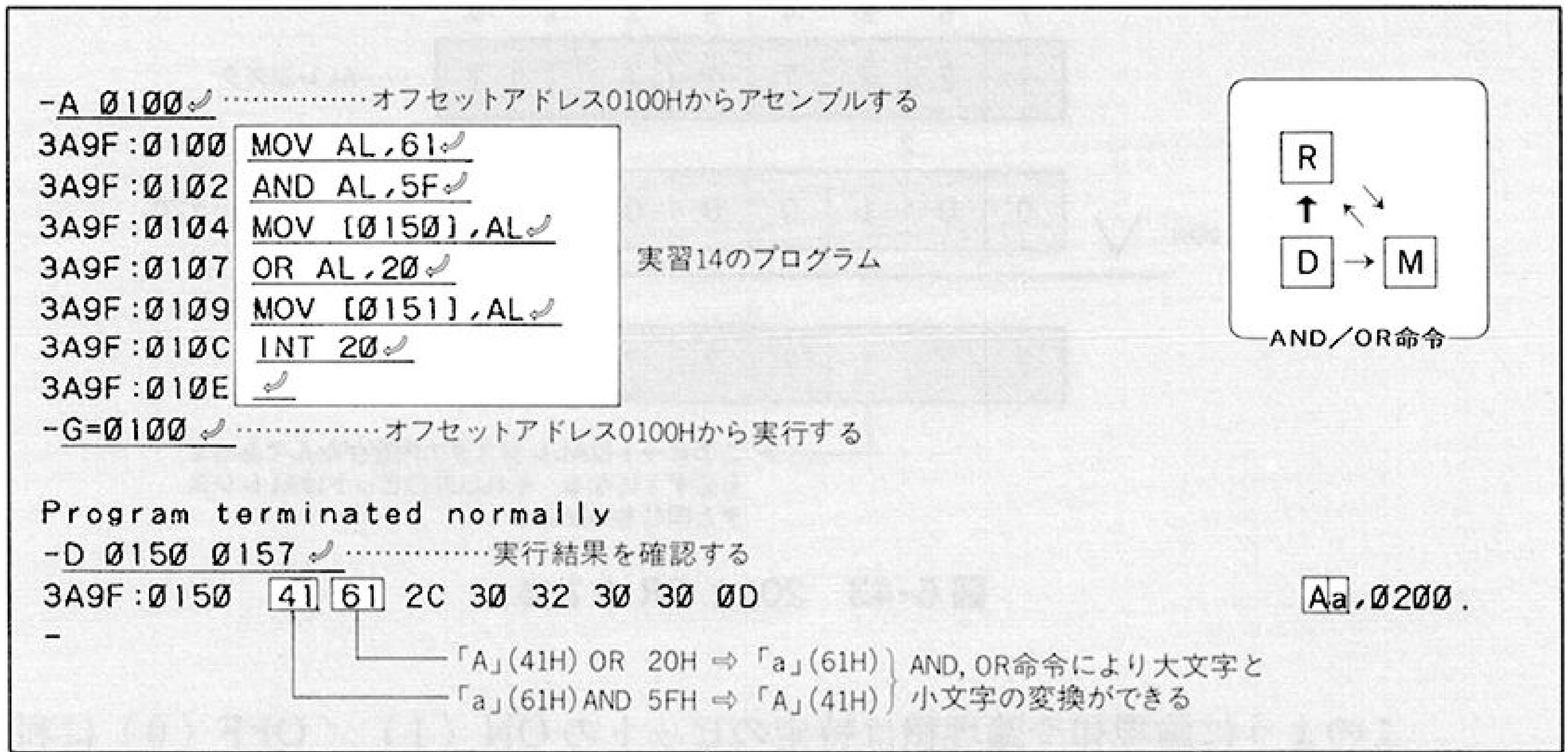


図 6-41　実習 14 の実行結果

この図からもわかるように，「$5F_H$」と AND をとることによって大文字に変換され，「20_H」と OR をとることによって小文字に変換されています．

このことをもっとくわしく見てみると，「$5F_H$」と AND をとるということは，第 5，第 7 ビットを 0 にするということを意味します．以下の図 6-42 を見てください．



図 6-42　$5F_H$と AND をとる

これに対して，「20_H」と OR をとるということは，第 5 ビットを 1 にするということを意味します（図 6-43）．

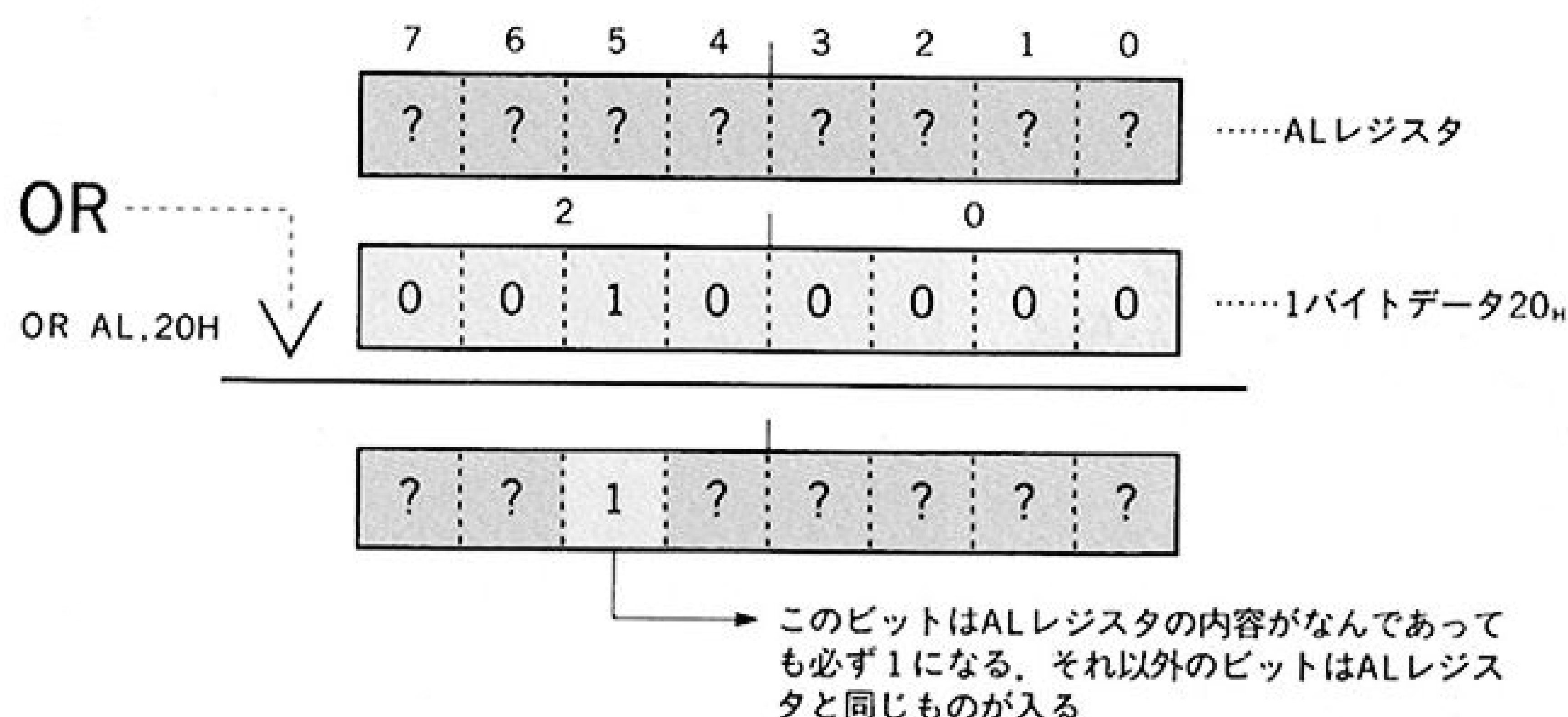
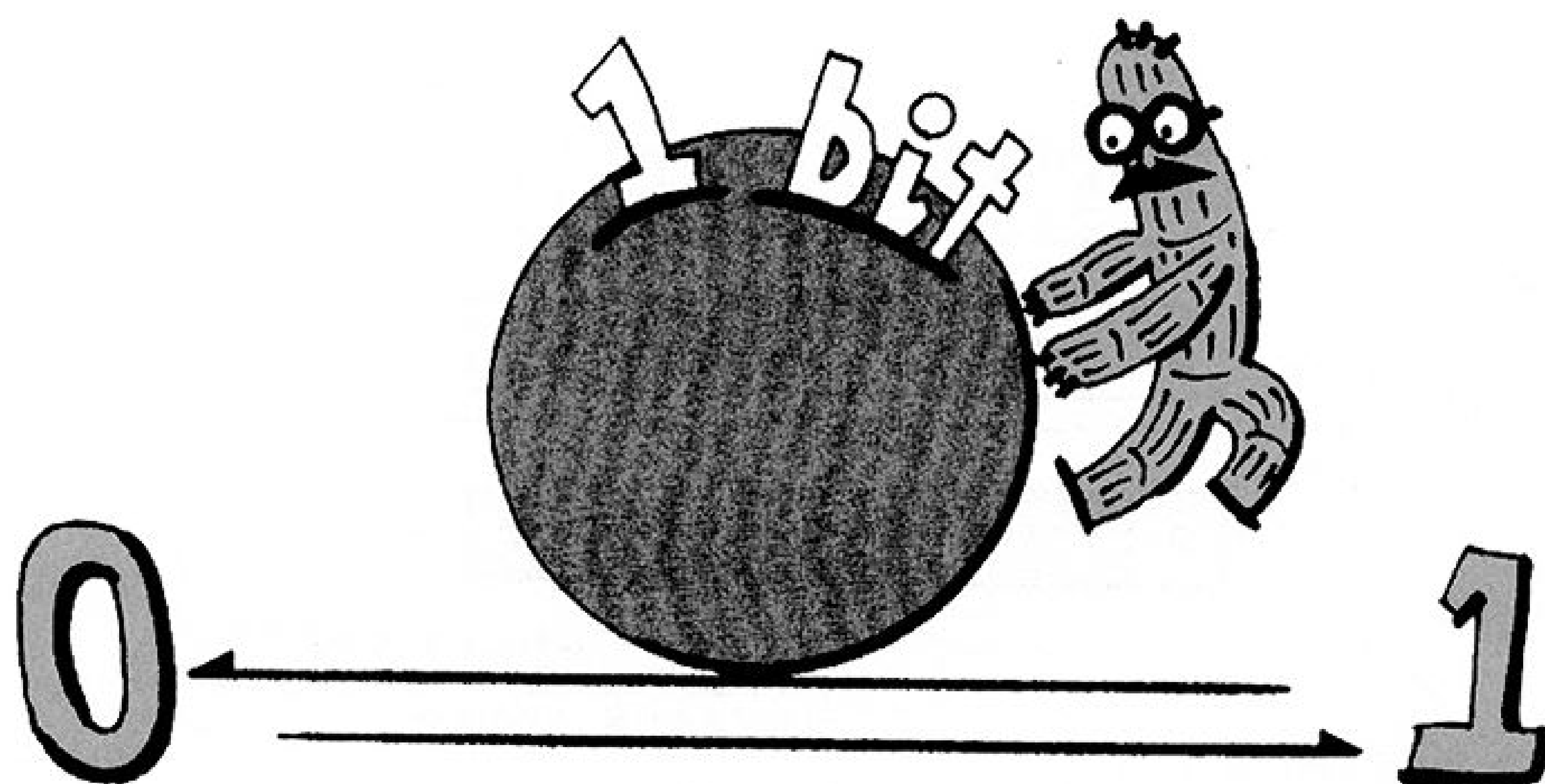


図 6-43 20Hと OR をとる

このように論理和や論理積は特定のビットのON（1）／OFF（0）に利用することができます．実習 14 のプログラムでは，アルファベットの大文字と小文字の位置関係が常に 20H (00100000) だけ離れている（273 ページのキャラクタコード表を参照)ことを利用して，AND 命令，OR 命令を使って第 5 ビットを ON（小文字になる）／OFF（大文字になる）し，大文字と小文字の変換を実現したというわけです*.



* AND 命令では，最上位ビット（第 7 ビット）もマスクしているが，これは英数字のみを扱うコンピュータではキャラクタコードを 7 ビットで表し，最上位ビットを文字列の終わりや属性を示すために使用するという習慣があったためである．なお，DEBUG の D コマンドで右側に表示されている文字も，最上位ビットをマスクしたデータに対応する文字が表示されている(39ページ参照)．

6.8 ローテート，シフト命令

ローテートとシフト命令は，レジスタやメモリの内容を「ビットパターン」と考えて，それを右あるいは左に1ビットずらす命令です．ずらして押し出されたビットを反対側の空いたビットに返して円環のような形式にするのを「ローテート」と言い，押し出されたままずらす形式を「シフト」と言います．

代表的なローテートとシフト命令をまず図で解説しましょう．なお，図では8ビットの場合しか解説しませんが，16ビットでもまったく同様に考えることができます．

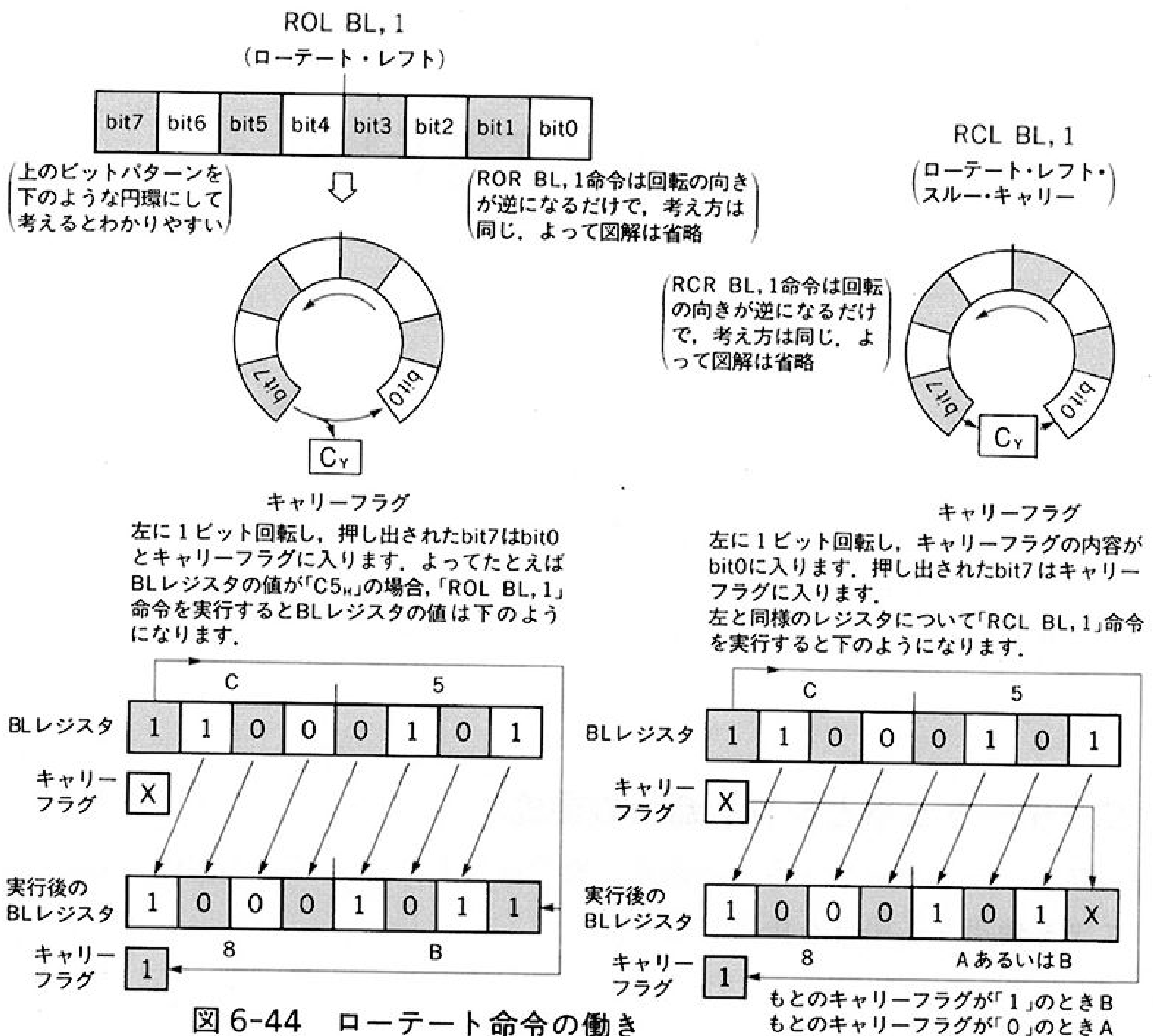


図6-44　ローテート命令の働き

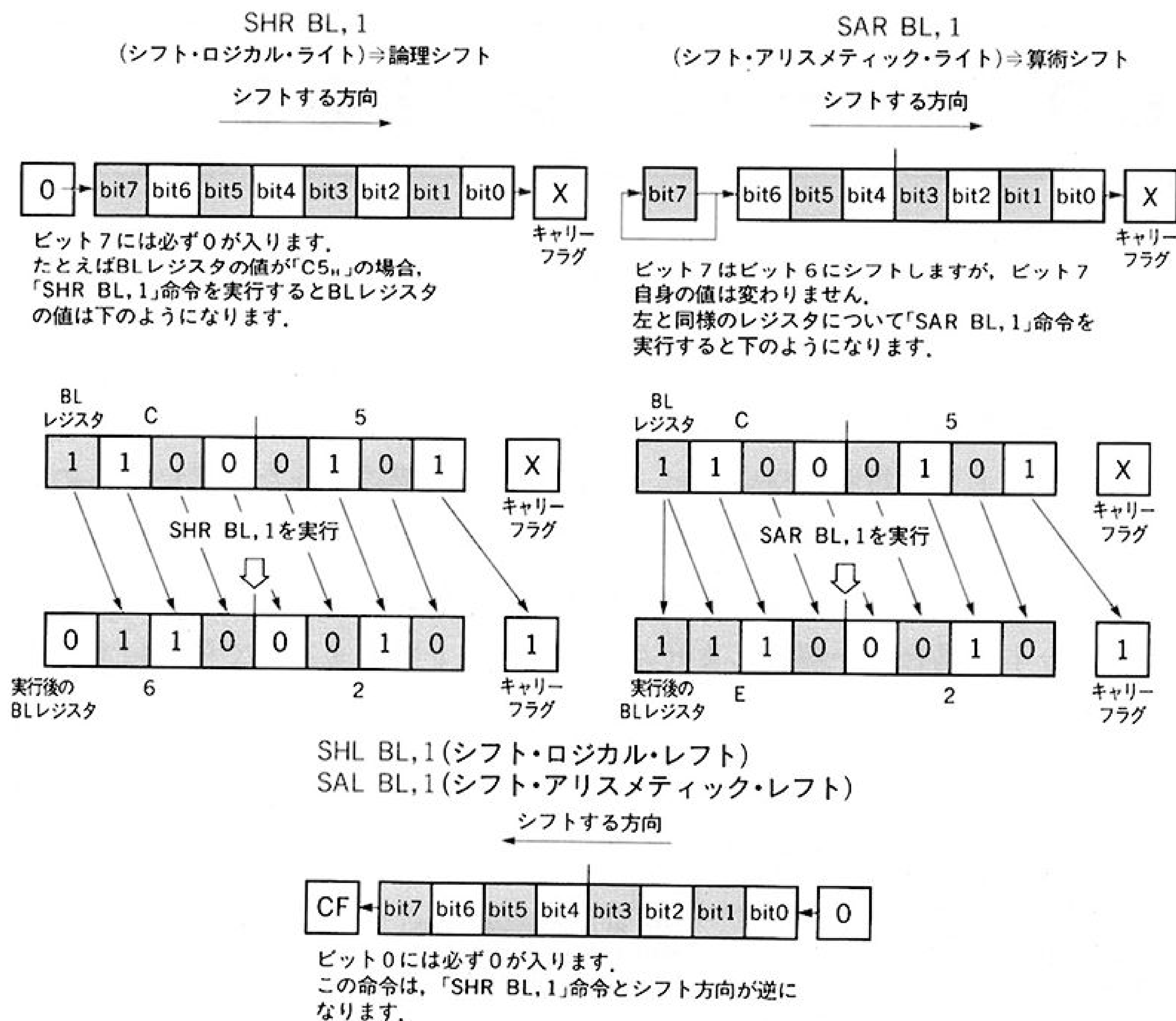


図6-45　シフト命令の働き

ローテート, シフト命令は, このようにいろいろな種類がありますが, これらの命令はさまざまなことに応用されています. たとえば, 「ローテートやシフトをしながらキャリーフラグを見ていれば, 何番目のビットが0であるか1であるかを判別できる」ということや, 「左にシフトすると値が2倍になり, 右にシフトすると値が1/2になる」というようなことに利用できます.

ローテート命令とシフト命令の書式

ローテートやシフト命令の書式を表で示しましょう. さきと同様にBLレジスタを使用する場合を例にします.

	左ローテート	右ローテート
キャリアを経由する	Rotate Left through Carry RC L　BL, 1	Rotate Right through Carry RC R　BL, 1
キャリアを経由しない	ROtate Left RO L　BL, 1	ROtate Right RO R　BL, 1

・「,1」は1ビットローテートするという意味がある

表6-9　ローテート命令の書式

	左シフト	右シフト
算術シフト	Shift Arithmetic Left SA L　BL, 1	Shift Arithmetic Right SA R　BL, 1
論理シフト	SHift logical Left SH L　BL, 1	SHift logical Right SH R　BL, 1

・「,1」は1ビットシフトするという意味がある

表6-10　シフト命令の書式

この表にある「論理シフト」と「算術シフト」の違いは，右シフトの場合に最上位ビットに0が入るか，最上位ビットが変化しないかということです(図6-45参照)．論理シフトではビットパターンを単純にシフトするだけですが，算術シフトではビットパターンを符号付きの数値として考え，シフトして値を1/2にしても符号が変わらないようにするのです．数値を符号付きと考えた場合には最上位ビットは符号ビットですから(65ページ参照)，このビットを変化させなければ符号が変わらないことがわかるでしょう．

左シフトの場合は論理的でも算術的でも同じですが，アセンブリ言語のニーモニックはその両方が用意されています．

●

ローテート，シフトの回数

ローテート，シフト命令で，最後に「,1」を付けるのは1ビットだけ，すなわち1回だけローテート，シフトするという意味です．ここで「1」以外の回数を直接指定することはできません*．

* 8086CPUの上位バージョンである80186，80286CPU，およびV30CPUでは，任意の回数を指定することができる（例：ROR BL,4)．

ただし，CL レジスタを使うと任意の回数を指定することができます．この場合は，

```
ローテートライト(ROtate Right)
ROR  BL , CL …… BL レジスタの内容を CL レジスタで指定する回数だけ右
                 にローテートする．
```

のような形式になります．「,1」のところを「,CL」に変えるだけで，CL レジスタで指定した回数だけローテート／シフトさせることができるのです．なお，CL レジスタの内容は変化しません．

実習 15 ROR 命令（ローテート・ライト）

ROR 命令の動作を確認する簡単な実習をしてみましょう．

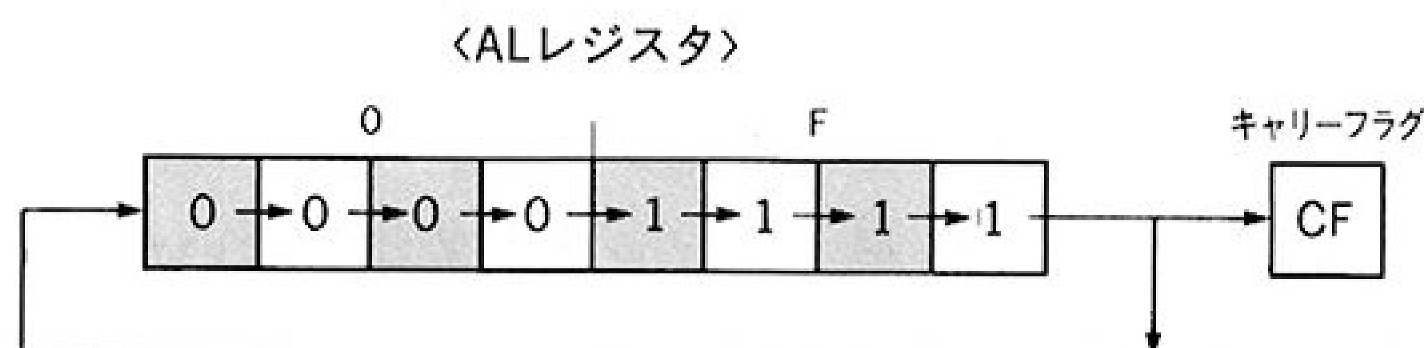


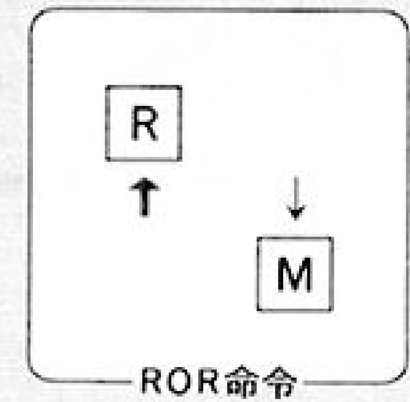
図 6-46 ROR 命令の動作

上のように「$0F_H$」のデータを AL レジスタにロードしておき，AL レジスタに対して ROR 命令を 8 回連続して実行します．DEBUG の T コマンドで 1 命令ずつ実行することによって，ローテートが行われる様子を確認しましょう．

ここでは T コマンドで命令数を指定して，マシン語命令を一度にトレースする方法を用います．その書式は次のようになります．

```
T<=オフセットアドレス> <命令数> … 指定したオフセットアドレス
                                 から指定した命令数だけマシ
                                 ン語命令をトレースする．
```

このプログラムは簡単なので，DEBUG の実行例のみを示します（図 6-47）．




```
-A 0100↵ ……………オフセットアドレス0100Hからアセンブルする
3716:0100 MOV AL,0F↵ ……………ALレジスタに0FHをセット
3716:0102 ROR AL,1↵
3716:0104 ROR AL,1↵
3716:0106 ROR AL,1↵
3716:0108 ROR AL,1↵
3716:010A ROR AL,1↵          ROR命令を8回繰り返す
3716:010C ROR AL,1↵
3716:010E ROR AL,1↵
3716:0110 ROR AL,1↵
3716:0112 ↵
-R↵ ……………レジスタの内容を確認する
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3716  ES=3716  SS=3716  CS=3716  IP=0100   NV UP EI PL NZ NA PO NC
3716:0100 B00F          MOV     AL,0F                                   0
-T=0100 9↵ ……………オフセットアドレス0100Hから9命令トレースする
AX=000F  |0|0|0|0|1|1|1|1|  0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3716      0       F      3716  CS=3716  IP=0102   NV UP EI PL NZ NA PO NC
3716:0102 D0C8          ROR     AL,1                                    0
AX=0087  |1|0|0|0|0|1|1|1|  0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3716      8       7      3716  CS=3716  IP=0104   OV UP EI PL NZ NA PO CY
3716:0104 D0C8          ROR     AL,1                                    1
AX=00C3  |1|1|0|0|0|0|1|1|  0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3716      C       3      3716  CS=3716  IP=0106   NV UP EI PL NZ NA PO CY
3716:0106 D0C8          ROR     AL,1                                    1
AX=00E1  |1|1|1|0|0|0|0|1|  0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3716      E       1      3716  CS=3716  IP=0108   NV UP EI PL NZ NA PO CY
3716:0108 D0C8          ROR     AL,1                                    1
AX=00F0  |1|1|1|1|0|0|0|0|  0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3716      F       0      3716  CS=3716  IP=010A   NV UP EI PL NZ NA PO CY
3716:010A D0C8          ROR     AL,1                                    1
AX=0078  |0|1|1|1|1|0|0|0|  0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3716      7       8      3716  CS=3716  IP=010C   OV UP EI PL NZ NA PO NC
3716:010C D0C8          ROR     AL,1                                    0
AX=003C  |0|0|1|1|1|1|0|0|  0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3716      3       C      3716  CS=3716  IP=010E   NV UP EI PL NZ NA PO NC
3716:010E D0C8          ROR     AL,1                                    0
AX=001E  |0|0|0|1|1|1|1|0|  0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3716      1       E      3716  CS=3716  IP=0110   NV UP EI PL NZ NA PO NC
3716:0110 D0C8          ROR     AL,1                                    0
AX=000F  |0|0|0|0|1|1|1|1|  0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3716      0       F      3716  CS=3716  IP=0112   NV UP EI PL NZ NA PO NC
3716:0112 BE0083        MOV     SI,8300                                 0
-
     └→ 8回シフトするともとに戻る
```

図 6-47　実習 15 のプログラムと実行結果

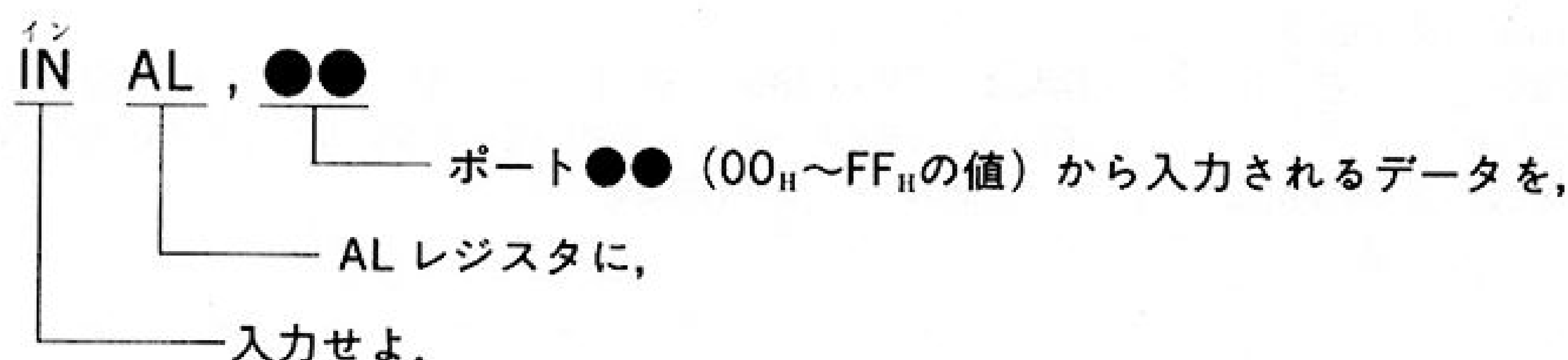
6.9 入出力命令

4.5 章で CPU の入出力に関して「ポート」や「ポートアドレス」を取り上げて解説しました．ポートに対して入力や出力を行うための命令が入出力命令です．ここでもう一度 81 ページの図 4-19 に目を通しておいてください．

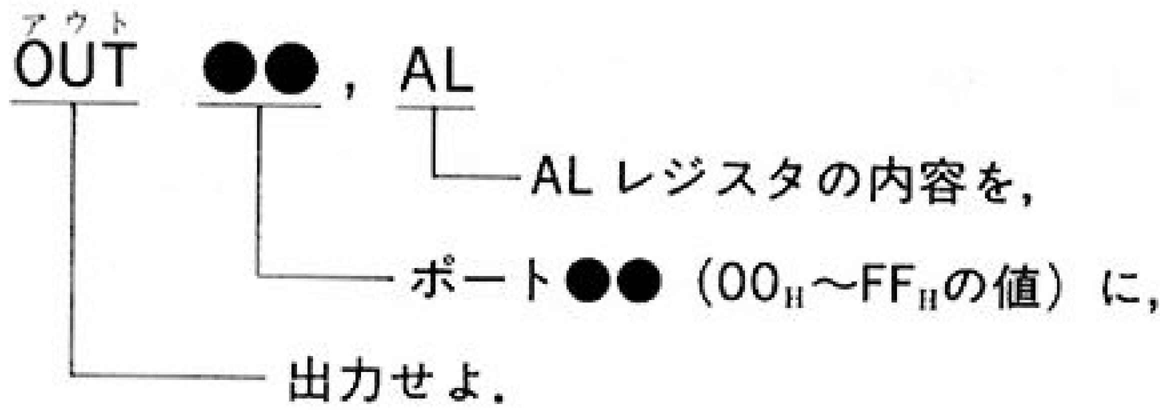
入出力ポートには，メインメモリ以外のすべての周辺装置が接続されていると考えてよいでしょう．キーボード，プリンタ，ディスク，RS-232C インターフェイスなどの外部の装置はもちろんのこと，CRT コントローラ，割り込みコントローラ，各種システムコントローラなどは，「入出力ポート」を介してデータバスやアドレスバスと結ばれているのです．そしてこれら外部の入出力装置は，入出力命令によってデータバスを通じて CPU とデータのやり取りを行います．

各周辺装置に接続されている入出力ポートも，メモリと同じようにアドレスを指定することによって，どの装置にアクセスするかを選択します．すなわちポートアドレスを指定するわけです．ポートアドレスとしては 16 ビットの値（65536 個のポートを選択できる）が使用されますが，そのうちの下位 8 ビット*（256 個）を数値で直接指定することができ，ここではその場合の実習を行います．

それでは命令の解説からはいっていきましょう．入力（IN）命令と出力（OUT）命令の書式を次に示します．



*上位 8 ビットは 0 になる．



「●●」として指定する値がポートアドレスです．この命令では8ビットの値で指定できるポートアドレスにALレジスタ（8ビット）の値を出力したり，8ビットの値をALレジスタに入力することができます．入出力命令もデータ転送命令の一種なので，データの転送方向はやはり右から左へ(←)と書きます．

なお，入出力命令ではポートに対するデータ転送の対象となるのは，ALレジスタかAXレジスタだけですから注意してください．また，ポートアドレスをこのように8ビットの値で指定するほかに，DXレジスタを使って16ビットのアドレスを指定する命令もありますが本書では扱いません．

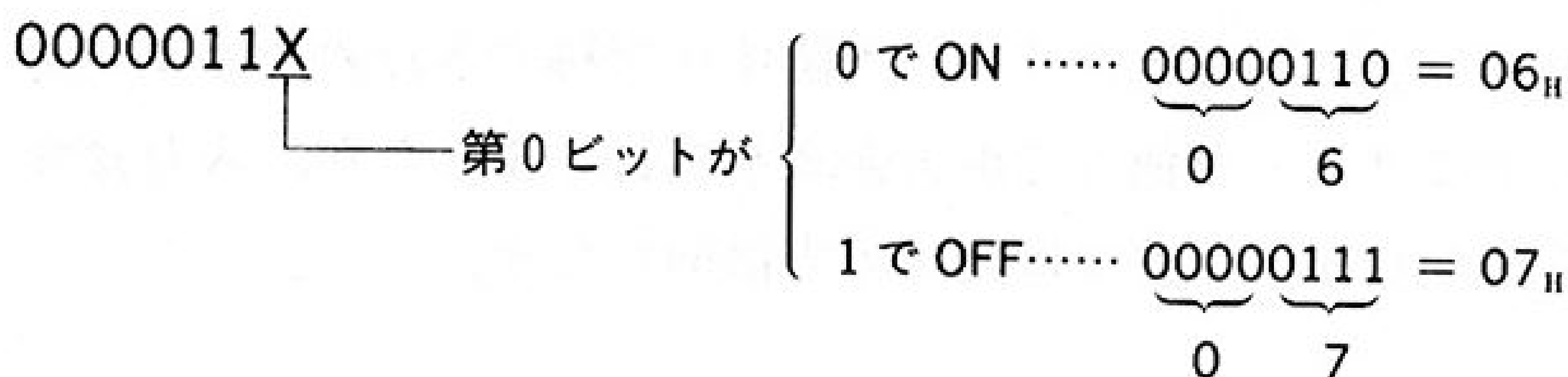
実習16　出力命令

それでは，入出力装置としてブザーを選び，「ポート」の概念を理解するために実験を行いましょう．

入出力装置のポートアドレスや，制御の仕方はハードウェアに固有のものです．つまり，機種に依存し，機種が異なればポートアドレスはもちろん制御の仕方も違うのが普通です．ここではPC-9801シリーズを例にします．

まず，ブザーがどのようにCPUに接続されているかを次ページの図6-48に示します．

ブザーのON／OFFはポートアドレス「37_H」のOUTポートに接続されています．ポートアドレス「37_H」はブザーを鳴らすためだけに用意されているわけではなく，ビット1と2が「1」のときにブザーは機能し，ビット0がブザーをON／OFFするスイッチとなります．



ブザーを鳴らすには，16 進で「06_H」という値をポート「37_H」に出力し，止めるには 16 進で「07_H」という値を同じポートに出力すればよいのです．

それでは DEBUG 上で実験してみましょう（図 6-49）．

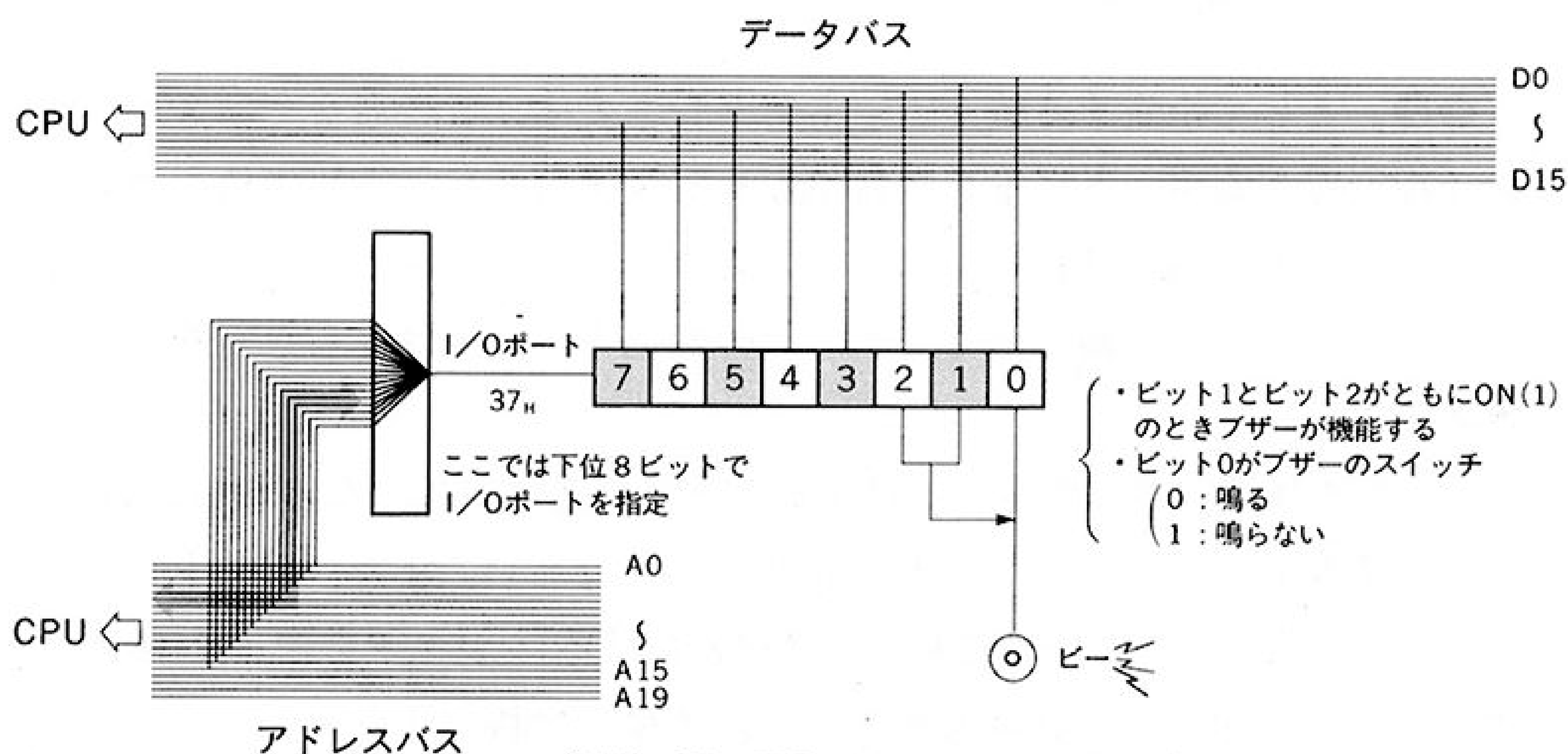


図 6-48　PC-9801 シリーズのブザーの入出力ポート

```
-A Ø1ØØ↵ ……………オフセットアドレス0100Hからアセンブルする
3A9F:Ø1ØØ  MOV AL,Ø6↵    ┐
3A9F:Ø1Ø2  OUT 37,AL↵    │ ブザーをON
3A9F:Ø1Ø4  ↵             ┘
-A Ø12Ø↵ ……………オフセットアドレス0120Hからアセンブルする
3A9F:Ø12Ø  MOV AL,Ø7↵    ┐
3A9F:Ø122  OUT 37,AL↵    │ ブザーをOFF
3A9F:Ø124  ↵             ┘
-G=Ø1ØØ Ø1Ø4↵ ……………オフセットアドレス0100Hから0104Hまでを実行する ⇨ ブザーが鳴り始める
AX=ØØØ6  BX=ØØØØ  CX=ØØØØ  DX=ØØØØ  SP=FFEE  BP=ØØØØ  SI=ØØØØ  DI=ØØØØ
DS=3A9F  ES=3A9F  SS=3A9F  CS=3A9F  IP=Ø1Ø4   NV UP EI PL NZ NA PO NC
3A9F:Ø1Ø4 A25ØØ1        MOV     [Ø15Ø],AL                    DS:Ø15Ø=41
-G=Ø12Ø Ø124↵ ……………オフセットアドレス0120Hから0124Hまでを実行する ⇨ ブザーが鳴りやむ
AX=ØØØ7  BX=ØØØØ  CX=ØØØØ  DX=ØØØØ  SP=FFEE  BP=ØØØØ  SI=ØØØØ  DI=ØØØØ
DS=3A9F  ES=3A9F  SS=3A9F  CS=3A9F  IP=Ø124   NV UP EI PL NZ NA PO NC
3A9F:Ø124 ØA4A42        OR      CL,[BP+SI+42]                SS:ØØ42=ØØ
-      〈注意〉この実習はPC-9801シリーズ用です
```

図 6-49　実習 16 のプログラムと実行結果

ここでもう一度図 6-48 を見てください．「ポート」とそれに接続されている入出力装置（この場合はブザー）の関係が理解できたのではないかと思います．ここではブザーを使って出力命令を実習しましたが，入力命令もデータの流れる方向が逆になるだけで考え方は同じです．

6.10　割り込み命令

割り込みはCPUの動作とは非同期に発生する外部の入出力要求に答えるために，プログラムの実行中に割り込んで実行される一種のサブルーチンです．コンピュータの入出力の実際を知るためにも，このことについて解説しましょう．

ハードウェア割り込み

MS-DOSの動作する多くのパソコンでは，キーボードの先行入力を受け付けます．たとえば，COPYコマンドなどを実行してフロッピーディスク装置が動作している間に「DIR ↵」と入力しておけば，COPYコマンドの実行が終了するとあたかもその時点で入力したかのように「DIR」という文字が表示され，実行されます．私たちはCOPYコマンドの実行終了を待つ必要はなく，実行中でも次のコマンドを入力することができるのです．このようなキーボードの先行入力を可能にしてくれるのが「割り込み」の機能です．

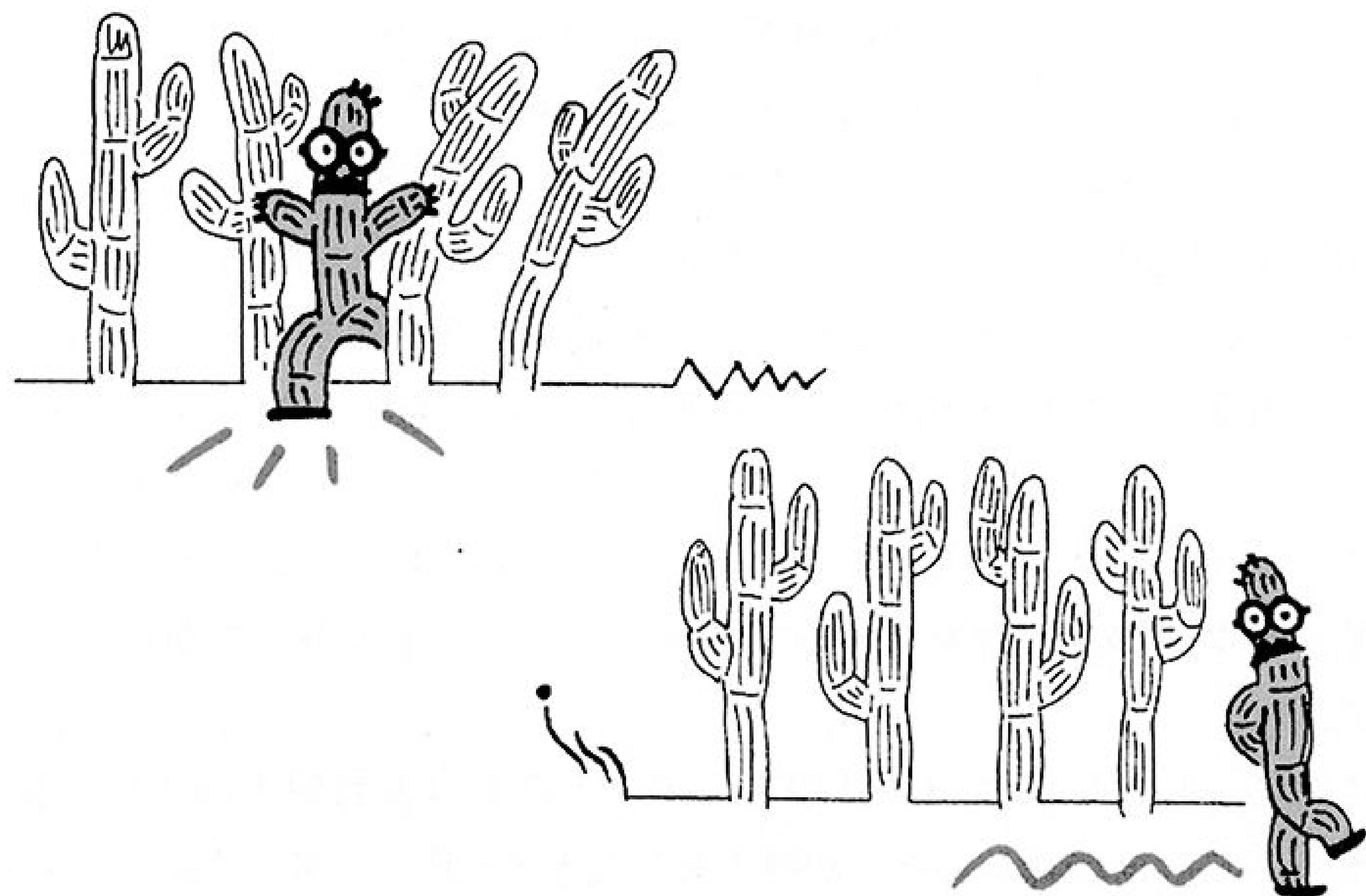
割り込みの機能がなければ，キーが押されたかどうかを調べるためにキーボードのI/Oをプログラムで監視している必要があります．ところがCOPYコマンドのように何かのプログラムを実行している間は，キーボードを監視しているプログラムは実行されていないので，キーボードが押されたことを検出することができず無視されてしまうことになります．先の例で言えば，COPYコマンドの実行が終了するまで次のコマンドを入力できないことになります．

そこで考えられたのが，CPUがキーボードのI/Oを監視するのではなく，キーボードのI/Oの側からキーが押されたことをCPUに知らせるというテクニックです．キーボードのI/Oはキーが押されると「割り込み信号」を専用の信号線を使ってCPUに送ります．CPUは割り込み信号を受け取るとそれ

まで実行していた処理を一時中断し，割り込み処理用のプログラム（割り込みルーチン）をサブルーチンとして実行します．このように，まさにプログラムの実行中に「割り込む」のです．I/O が CPU に割り込み信号を送ることを，「割り込みをかける」とも言います．

割り込みルーチンでは，キーボードの I/O を調べてどのキーが押されたかをメモリに記録しておきます．上の例においては COPY コマンドの実行が終了すると，次のコマンドを実行するためのキーボード入力ルーチンにおいて（キーボードの I/O ではなく）そのメモリの内容を調べ，それによってどのキーが押されたかを知るのです．この割り込みによる先行入力の原理を図解したのが図 6-50 です．

割り込みはキーボードだけでなく，RS-232C，ディスクドライブ，CRT コントローラ，マウスなどいろいろな I/O から発生します．CPU は各 I/O をプログラムで監視する代わりに割り込みを受け付けることで，入力の有無や入出力の終了を知るのです．このような I/O による割り込みを後で解説するソフトウェア割り込みと区別するために「ハードウェア割り込み」と呼びます．



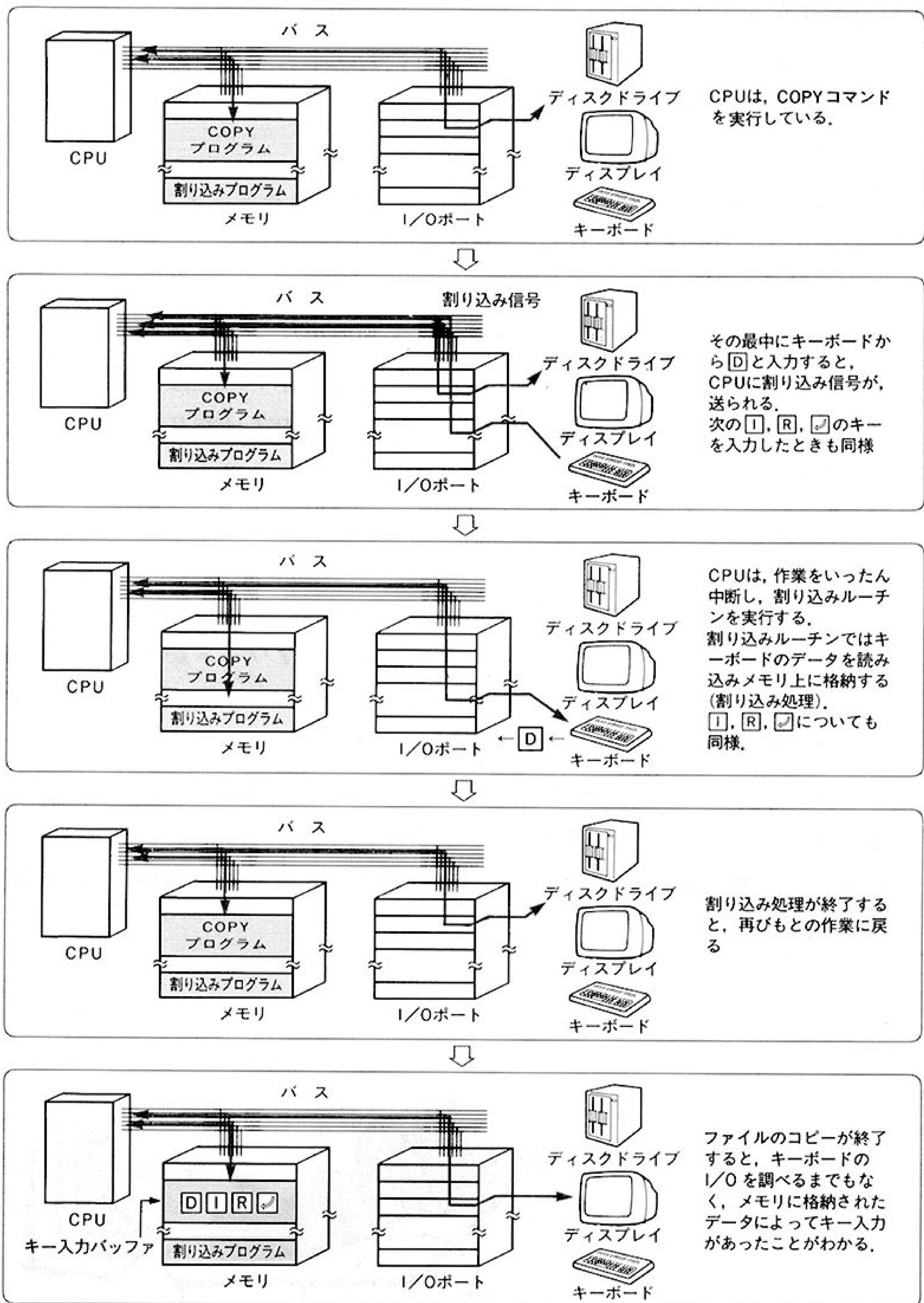


図 6-50　キーボードの先行入力の仕組み

割り込みベクタ

I/O から割り込みがかかると，その I/O に対応する割り込み処理ルーチンが実行されます．CPU はどうやって目的の割り込み処理ルーチンを選択し，その開始アドレスを知るのでしょうか？

割り込みは，割り込みが発生する装置に番号を持たせることで区別します．装置が割り込みをかけると，その番号が CPU に伝えられるような仕組みになっているのです．この番号を「割り込みタイプ」と呼びます．

割り込みによって実行される割り込みルーチンのアドレスは，「割り込みベクタ」と呼ばれます．割り込みベクタは割り込みタイプの順に「割り込みベクタテーブル」というところに格納されています．CPU は割り込みを受けるとその割り込みタイプにしたがって割り込みベクタテーブルから割り込みベクタを取り出し，そのアドレスから始まるルーチンをサブルーチンとして実行します．

割り込みベクタテーブルは，物理アドレス「00000_H」からの 400_H バイト（1K バイト）に固定的に割り当てられています．最初の 2 ワードがタイプ 0 の割り込みルーチンのセグメントアドレスとオフセットアドレス，次の 2 ワードがタイプ 1 のセグメントアドレスとオフセットアドレス……，というように順番に割り込みベクタが格納されています．

割り込みの発生により，割り込みベクタが割り込みベクタテーブルから取り出されて，割り込みルーチンが実行されるという仕組みを図解したのが図 6-51 です．



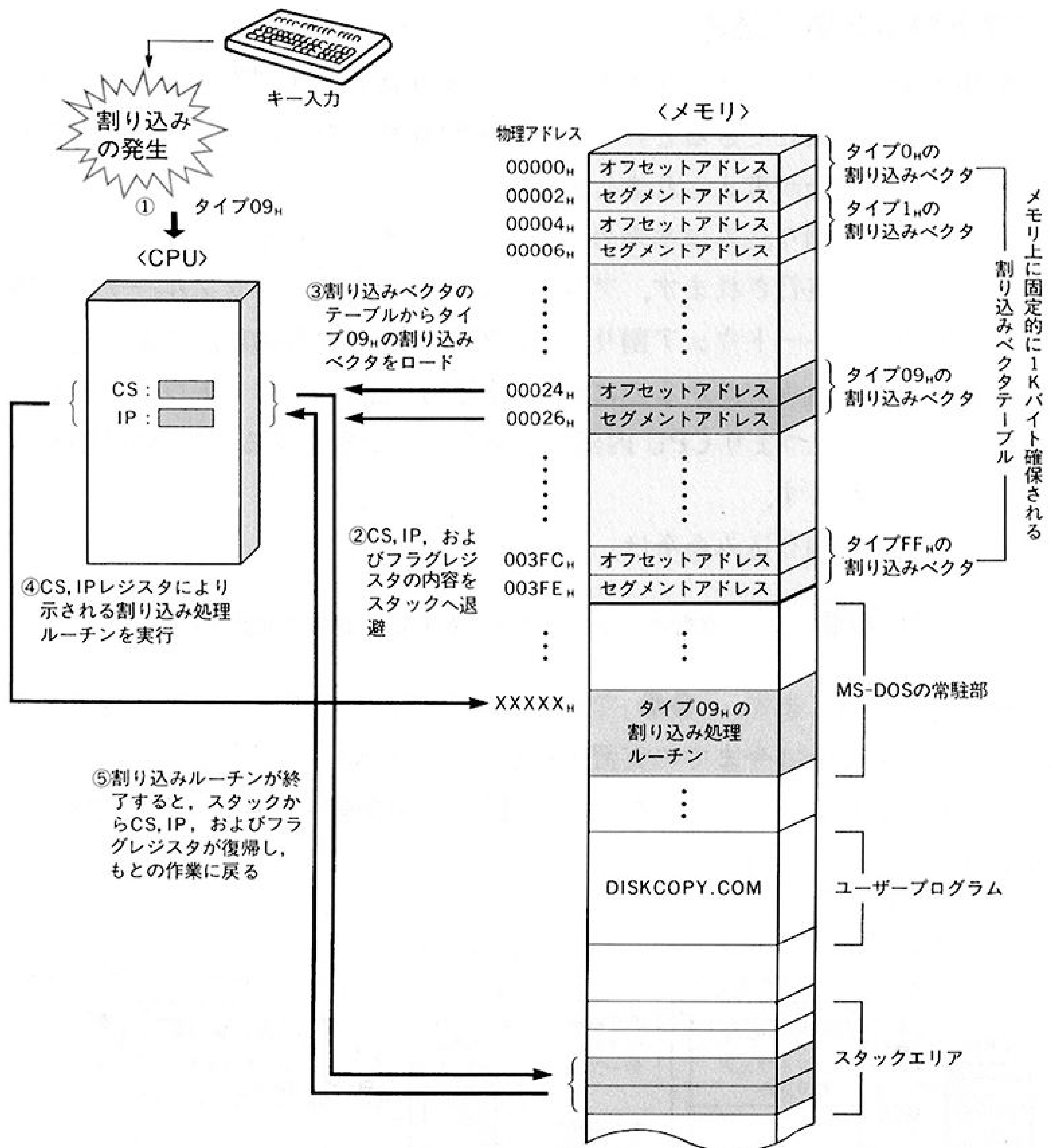


図 6-51　割り込みベクタと割り込み処理ルーチンの実行手順

ソフトウェア割り込み

割り込みベクタテーブルを利用すると，割り込みタイプを指定するだけでサブルーチンを実行できるという点で非常に便利なため,「ソフトウェア割り込み」という概念が考えられました.

ソフトウェア割り込みは本来のハードウェア割り込みとは異なり，マシン語命令によって実行されます．プログラムの実行方法は，サブルーチンの場合と同様です．ハードウェア割り込みが外部の装置から物理的な割り込み信号によって割り込みがかかるのに対し(外部割り込み)，ソフトウェア割り込みはプログラム，つまりCPU内部から割り込みが発生するので「内部割り込み」とも呼ばれます.

ソフトウェア割り込み命令は,

INT（インタラプト(INTerrupt)） **●●** …… **●●番のソフトウェア割り込み命令を実行する.**

という形式で表します.「●●」で指定する1バイトのデータが割り込みタイプです．この命令は今までの実習のなかで何度か使ってきました.

ハードウェア割り込みとソフトウェア割り込みの違いを図6-52に図解しましょう.

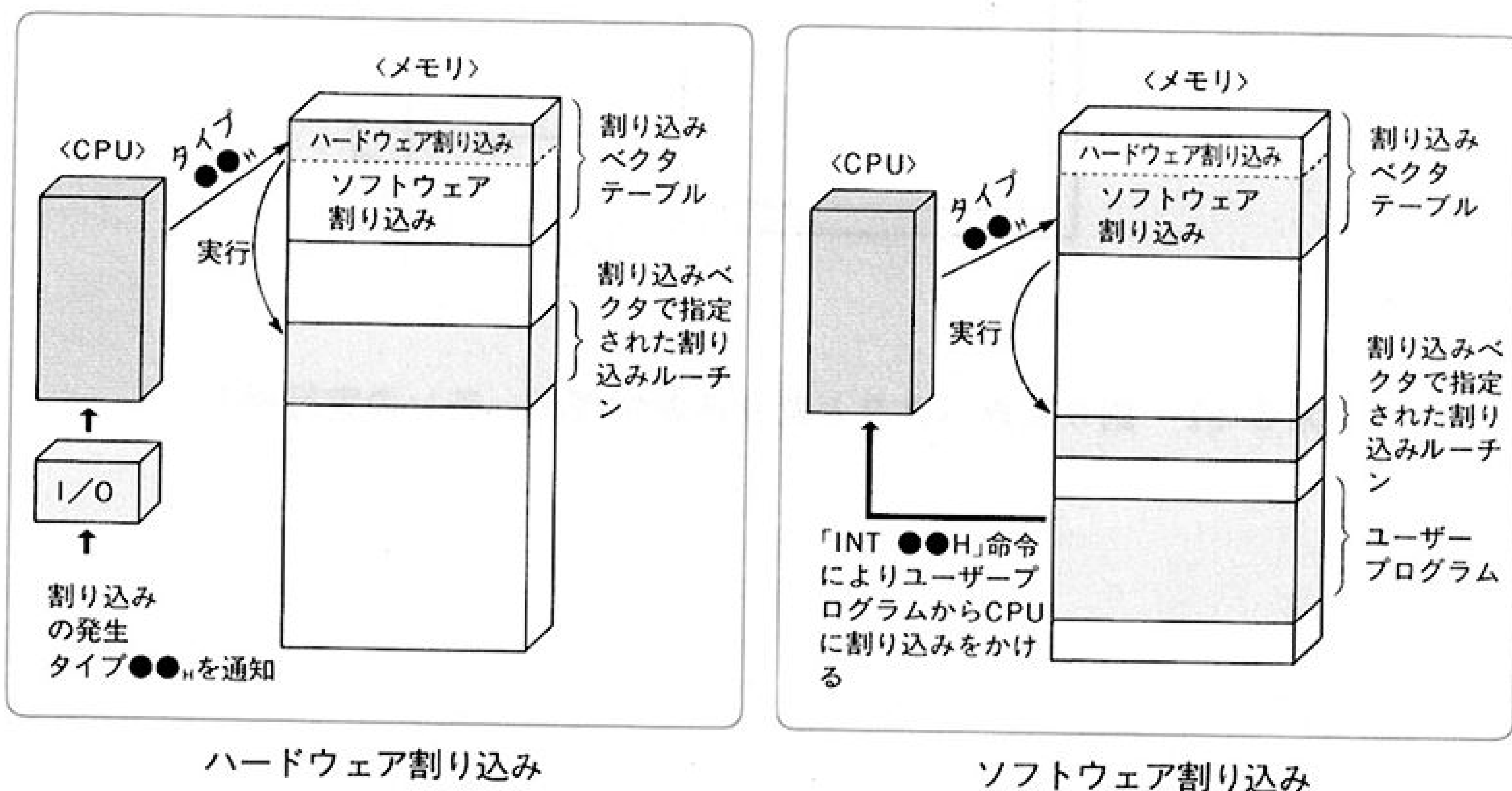


図6-52　ハードウェア割り込みとソフトウェア割り込み

システムコール

MS-DOSの「システムコール」は，上で解説したソフトウェア割り込みを利用しています．MS-DOSにはコンソールやディスクとの基本的な入出力を提供（サービス）するという役割がありますが，その機能を呼び出すことをシステムコールと言います．このようなMS-DOSのサービスルーチンの機能はどの機種でも同じですが，メモリ上に実際に配置されているアドレスは機種によって異なり，MS-DOS上で実行されるプログラムからはわかりません．つまり，コール命令を使って呼び出すわけにはいかないのです．そこでソフトウェア割り込みを利用します．MS-DOSの各種の機能を呼び出すための割り込みタイプを共通に固定しておけば，機種が違っても同じ方法で呼び出すことができます．

割り込みタイプ20_Hのソフトウェア割り込みは，「プログラム終了」のシステムコールです．本書の実習のように，DEBUG上でプログラムを入力している場合や，それを実行型ファイルへ書き出して実行する場合には，このシステムコールを呼び出すことによってプログラム終了時に必要な各種の処理を行い，DEBUGのプロンプトやMS-DOSのコマンド待ち状態に戻ることができます．

割り込みタイプ21_Hのソフトウェア割り込みは，他のシステムコールと異なり，その中が多くの機能（ファンクション）に分かれています．このため，このシステムコールを特に「ファンクションコール」と呼びます．このファンクションコールを呼び出すことにより，MS-DOSのさまざまな機能を利用することができます．

ファンクションコールでは，AHレジスタにファンクション番号を入れて目的の機能（ファンクション）を指定します．つまり，

```
MOV  AH ,○○H  }  MS-DOSのファンクション○○の機能をソフトウェ
INT  21H      }  ア割り込みにより呼び出す.
```

という命令を実行することによって呼び出します．呼び出す機能によってはALレジスタやDXレジスタを使って割り込みルーチンへ必要なパラメータを渡します．その結果はレジスタやメモリに返されるか，動作となって現れます．私たちはこの簡単な命令によって，MS-DOSのサービスルーチンのアド

レスやそこで何が行われているかを知らなくても，ファンクションの番号とその機能を知っていれば，これまで使ってきた１文字入力や文字列出力を始めとするMS-DOSの提供する多くの有用なサービスを受けることができるのです．

この仕組みを図解したのが図 6-53 です．

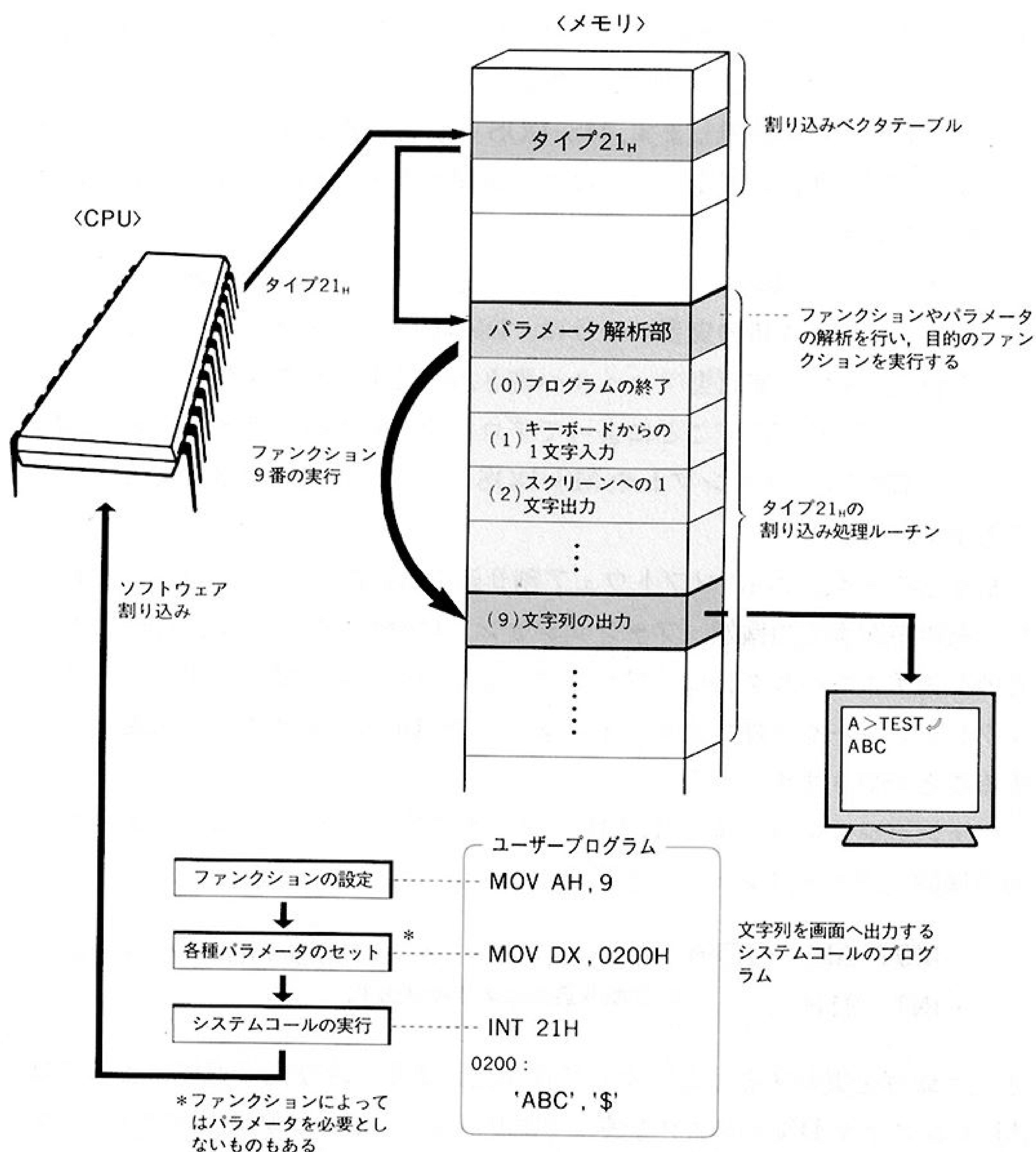


図 6-53　ファンクションコールの仕組み

MS-DOSのファンクションコールは約70種類ありますが*，以下では本書で使用した基本的なファンクションとその使い方を**表6-11**に挙げておきます．これらのファンクションコールは7章のサンプルプログラムでも使われていますから，具体的な使い方は次章を参照してください．

ファンクション番号	機　能	パラメータ	使用例
01_H	キーボードからの1文字入力とエコー表示	入力された文字をALレジスタに返す	MOV AH, 01H INT 21H
02_H	スクリーンへの1文字出力	DLレジスタの内容をスクリーンに表示する	MOV AH, 02H MOV DL, [キャラクタコード] INT 21H
06_H	コンソールからの直接入力（[CTRL]+[C]などのチェックを行わない．また入力時には文字がスクリーンにエコーされない）	DLレジスタに「FF_H」をセットすると入力*．それ以外ではその文字をスクリーンに出力する	MOV AH, 06H MOV DL, [XXH] INT 21H
09_H	スクリーンへの文字列出力	DXレジスタの内容をオフセットアドレスとするメモリから連続するメモリの内容を文字列として出力する．文字列の最後は"$"で指定する	MOV AH, 09H MOV DX, [XXXXH] INT 21H
$0A_H$	コンソールからの文字列入力（バッファード・キーボード入力）	キーボードから1行入力し，DXレジスタの内容をオフセットアドレスとするメモリに格納する**	MOV AH, 0AH MOV DX, [XXXXH] INT 21H

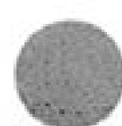
＊入力の場合，入力がなければゼロフラグがセットされ，入力があればゼロフラグがリセットされる．また入力された文字はALレジスタに返る．くわしい使い方は「7.1 スロットマシーン・プログラム」で解説する．
＊＊文字列を格納するバッファの構造など，くわしい使い方は「7.2 文字列反転プログラム」で解説する．

表6-11　本書で使用したファンクションコールの一覧

＊これはversion2.11の場合．version3.1では約90種類ある．なお，ファンクションコールについてのくわしい解説は，「応用MS-DOS」（アスキー発行）などの他の書籍を参照するとよい．

7

やさしいプログラミングの実　　例



前章までの解説をもとに，実際にプログラミングを体験してみましょう．6章では主要なマシン語命令を実習してきましたが，その1つ1つは非常に単純な機能しか持っていません．それらをいかに組み合わせて目的のあるプログラムを作成するかが，マシン語プログラミングのポイントです．ここでは，なるべく簡単で興味深い例題を選びましたので，自分がプログラムを作成しているつもりで実習をしてみてください．また，できればこれまでの知識を応用してここでのプログラムの拡張に挑戦してくれることを期待します．

わからない部分やわからない命令があれば，適宜これまでの関連する章を読み返してみるとよいでしょう．

7.1 スロットマシーン・プログラム

非常に簡単なサンプルプログラムからプログラミングの実習を始めましょう．この節で取り上げるのは，「スロットマシーン・プログラム」です．スロットマシーンといっても3つのドラムがくるくる回って，というおおげさなものではなく，0から9までの数字が目まぐるしく表示され，何かキーを押すとそこで表示が止まるというものです．ちょっとしたくじや，乱数に使うこともできるでしょう．

プログラムの流れは次のようになります．

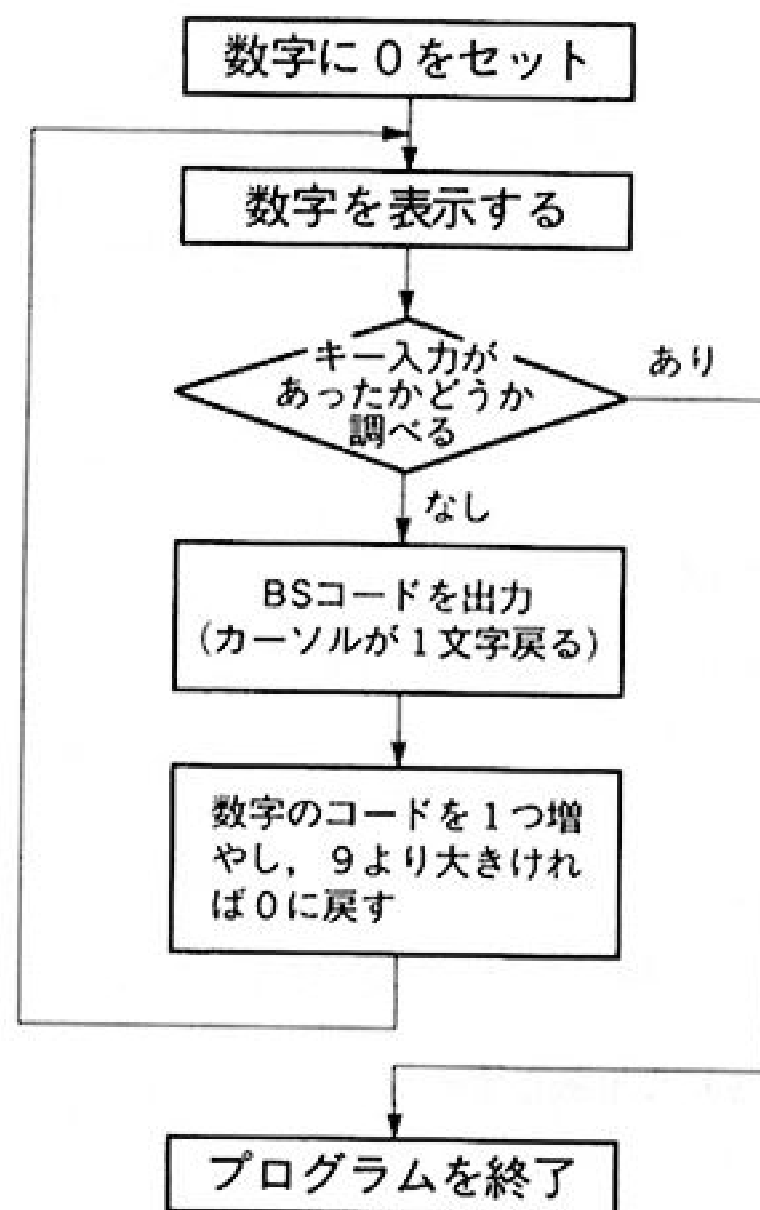


図7-1　スロットマシーン・プログラムの流れ

このプログラムでは，画面上に文字を表示したり，キー入力を調べるために，「6.10 割り込み命令」のところで取り上げた2番と6番の2つのファンクションコールを使用します（219ページの表6-11を参照）．

6章の実習10で取り上げたファンクションコール1番では，キーボードからの入力があるまでそのまま待つのですが，ここで用いるファンクションコール6番で入力を行うと，入力がない場合にはそれを待たずに帰ってきます．また，画面へのエコーや[CTRL]+[C]のチェックも行いません．

それでは，実際にプログラムを作成し実行してみましょう．

```
A>DEBUG
-A 0100 ……オフセットアドレス0100H番地からアセンブルする
3633:0100 MOV BL,30 ……BLレジスタに'0'のキャラクタコード(30H)をロード
3633:0102 MOV AH,02 ┐
3633:0104 MOV DL,BL │ BLレジスタの内容をコンソールへ1文字出力(システムコール2番)
3633:0106 INT 21    ┘
3633:0108 MOV AH,06 ┐
3633:010A MOV DL,FF │ コンソールから1文字入力(システムコール6番)
3633:010C INT 21    ┘
3633:010E JNZ 011F ……入力があればオフセットアドレス011FH番地へジャンプ
3633:0110 MOV AH,02 ┐
3633:0112 MOV DL,08 │ BS(バックスペースのキャラクタコード08H)をコンソールへ
3633:0114 INT 21    ┘ 1文字出力(システムコール2番)
3633:0116 INC BL ……BLレジスタの内容をインクリメント
3633:0118 CMP BL,39 ……BLレジスタの内容と'9'のキャラクタコード(39H)を比較
3633:011B JA 0100 ……BLレジスタの内容が'9'より大きければオフセットアドレス0100H番地へジャンプ
3633:011D JMP 0102 ……オフセットアドレス0102H番地へジャンプ
3633:011F INT 20 ……プログラムの終了
3633:0121
-N SLOT.COM ……コマンド名をセットする
-R BX   ┐
BX 0000 │
:0000   │ BX：CXレジスタに書き込むバイト数をセットする
-R CX   │     BX ← 0000H
CX 0000 │     CX ← (プログラムの最終アドレス) − 0100H
:0021   ┘
-W ……ディスクへ書き込む
Writing 0021 bytes
-Q

A>DIR SLOT.COM ……コマンドが作成されたかどうかを確認する

 ドライブ A: のディスクのボリュームラベルはありません
 ディレクトリは A:¥

SLOT     COM        33  86-12-04  22:56
        1 個のファイルがあります
   482304 バイトが使用可能です

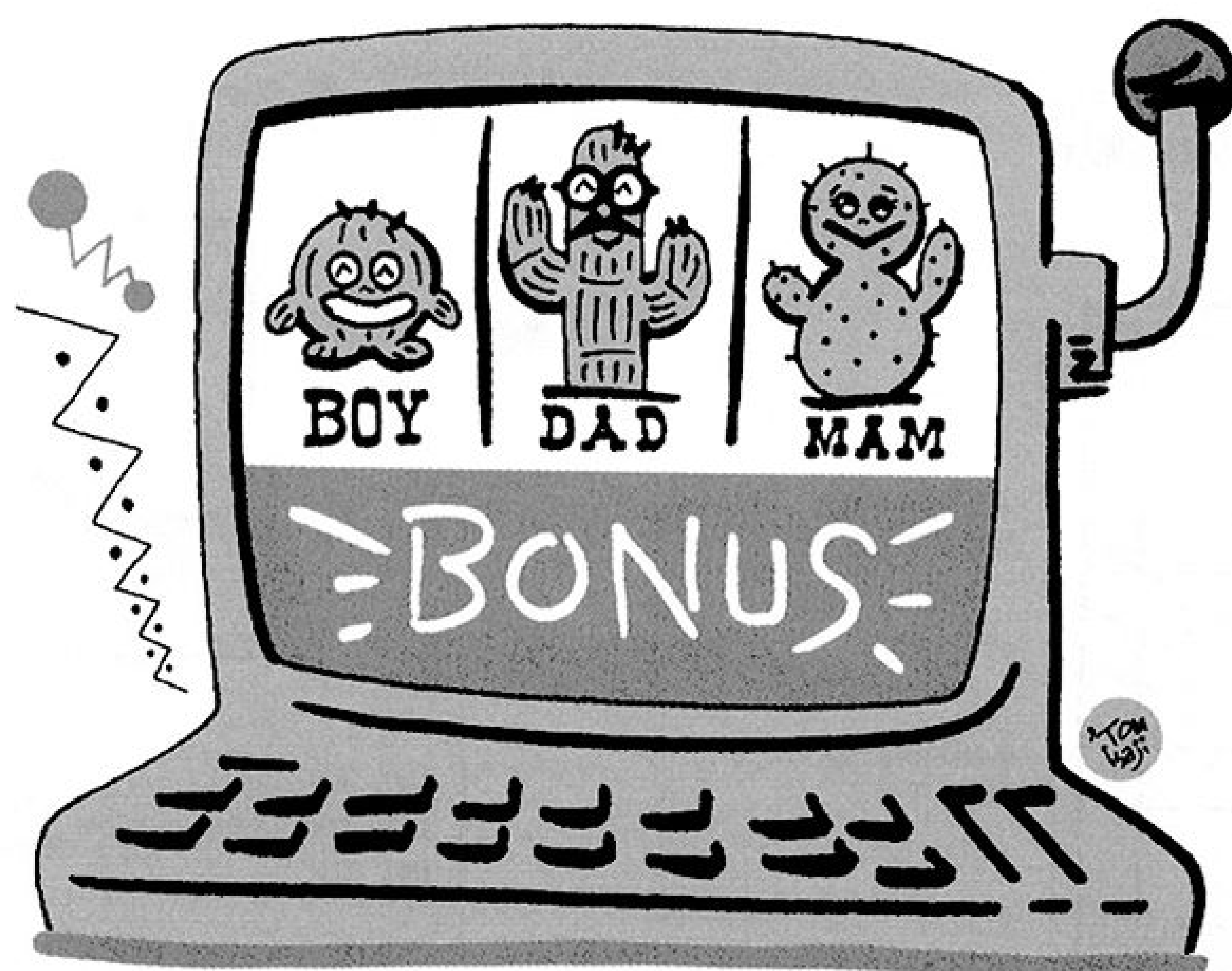
A>SLOT ……実行してみる
0 ……0～9までの数字がくるくると表示され，何かキーを押すと止まる
A>
```

図7-2　スロットマシーン・プログラムと実行例

そうむずかしいプログラムではないので，プログラムの流れを理解することはたやすいと思います．個々の命令について不明な点があればもう一度6章を見てください．このプログラムは，6章のコラム「MS-DOSの実行型ファイルを作成するには…」で取り上げた方法でプログラムをディスクに書き込み，MS-DOSの実行型ファイルとして実行しています．

また，「MOV BL,30」として最小の数字〝0〞のキャラクタコードをロードし，「CMP BL,39」として最大の数字〝9〞のキャラクタコードとの比較を行っているところを，他の値，たとえば〝1〞と〝6〞のキャラクタコードである「31_H」と「36_H」にそれぞれ変更するとサイコロにすることもできます．必要に応じて適宜変更して楽しんでください．

なお，DEBUGではすべての数値が16進数として扱われるので「H」を付けませんが，次章で取り上げるMASMのプログラムでは16進の数値に「H」を付けなければならないことに注意してください．

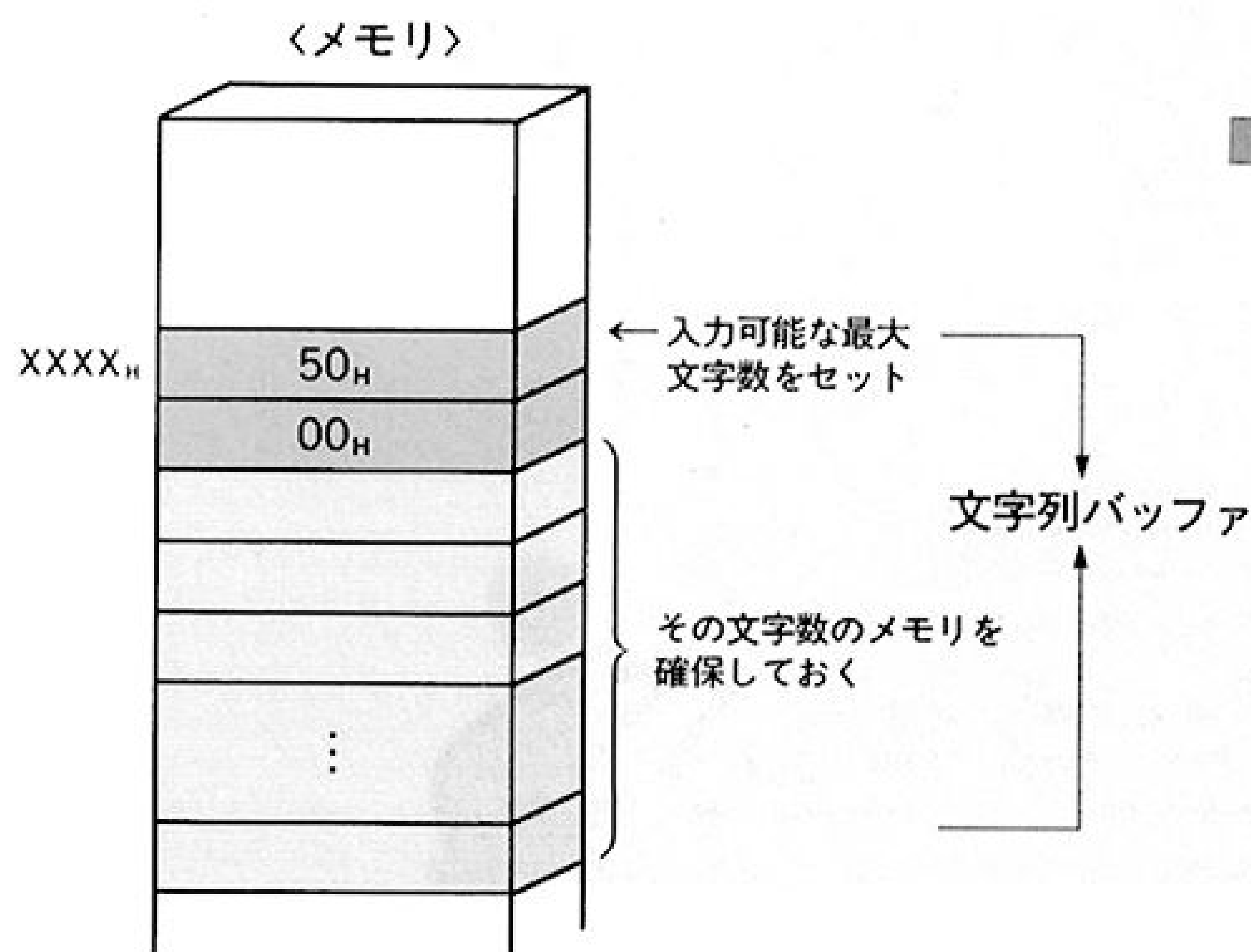


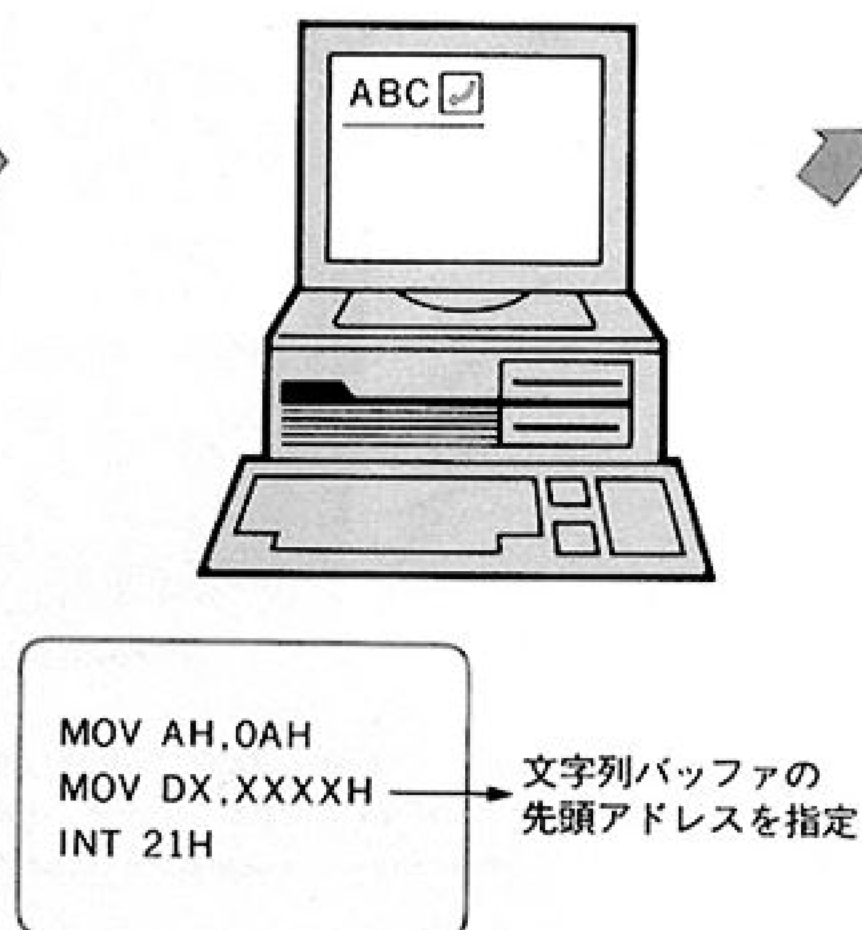
7.2 文字列反転プログラム

次に作成するのは，文字列を反転させるプログラムです．コンソールから文字列を入力し，文字の順番を逆さまにして表示します（実行例を参照してください）．

このプログラムではファンクションコール$0A_H$番を用いて，コンソールから文字列を入力し，入力された文字列をDXレジスタの内容をオフセットアドレスとするメモリに格納します．文字列を格納するバッファは以下の図7-3のような構造をしており，バッファの1バイト目にあらかじめ入力可能な文字数をセットして，その数だけ領域を確保しておかなければなりません．

文字列が入力されると，入力された文字数がバッファの2バイト目に返され，続くアドレスに入力された文字列が1文字ずつ格納されます．文字列を反転して表示するためには，この文字列をおしまいから1文字ずつ表示していけばよいわけです．





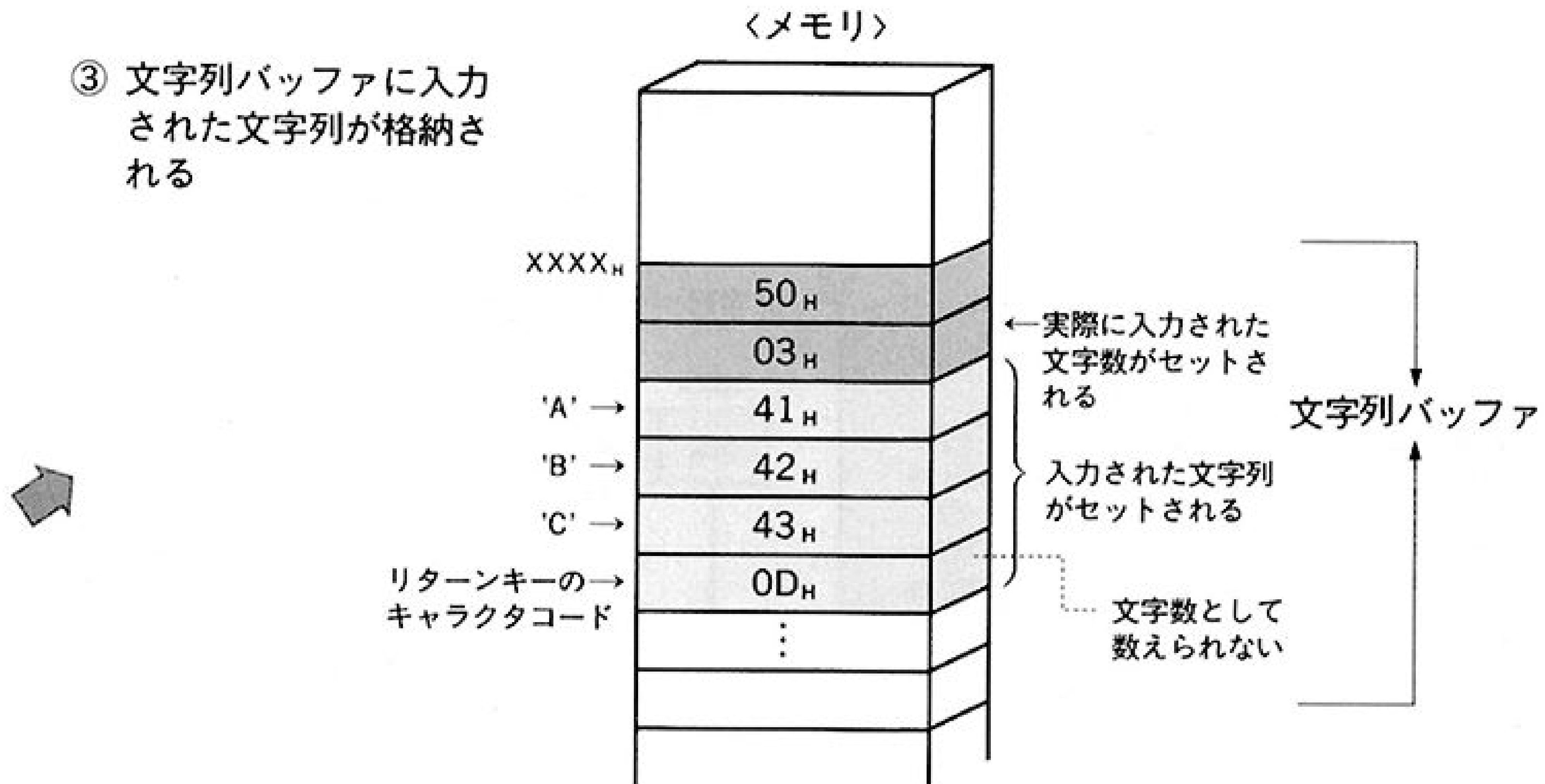


図7-3　ファンクションコール0AH番で使用する文字列バッファの構造

それでは実際にプログラムを入力し，実行してみましょう．

```
A>DEBUG↵
-A 0100↵
3CFB:0100 MOV AH,0A          } コンソールから1行入力．DXレジスタに文字列
3CFB:0102 MOV DX,0132        } バッファのアドレスをセット ←(B)
3CFB:0105 INT 21             } (ファンクションコール0AH番)
3CFB:0107 XOR BH,BH          ……BHレジスタを0にする(なぜこうなるか考えてみてください→198ページ参照)
3CFB:0109 MOV BL,[0133]      ……BLレジスタに入力された文字数をロード
3CFB:010D CMP BL,00          ……BLレジスタの内容と0Hを比較
3CFB:0110 JZ 012D            ……BLレジスタが0Hならば入力がなかったので，012DH番地へジャンプ
3CFB:0112 MOV AH,9           } コンソールへの文字列出力．DXレジスタに改行
3CFB:0114 MOV DX,012F        } 文字列のアドレスをセット(ファンクションコー
3CFB:0117 INT 21             } ル09H番)→7.3節で解説
3CFB:0119 MOV AH,02          } コンソールへ1文字出力．BL＋0133H番地の
3CFB:011B MOV DL,[BX+0133]   } メモリの内容をDLレジスタにセット ←
3CFB:011F INT 21             } (ファンクションコール02H番)
3CFB:0121 DEC BX             ……BXレジスタの内容をデクリメント
3CFB:0122 JNZ 0119           ……0でなければ0119H番地へジャンプ
3CFB:0124 MOV AH,09          } コンソールへの文字列出力．DXレジスタに改行
3CFB:0126 MOV DX,012F        } 文字列のアドレスをセット(ファンクションコー
3CFB:0129 INT 21             } ル09H番)→7.3節で解説
3CFB:012B JMP 0100           ……プログラムの先頭(0100H番地)へジャンプ
3CFB:012D INT 20             ……プログラムの終了 ←
3CFB:012F DB 0D,0A,'$'       ……改行文字列のセット ➡(A)
3CFB:0132 DB 50,0            ……文字列バッファの文字数をセット ➡(B)
3CFB:0134
```

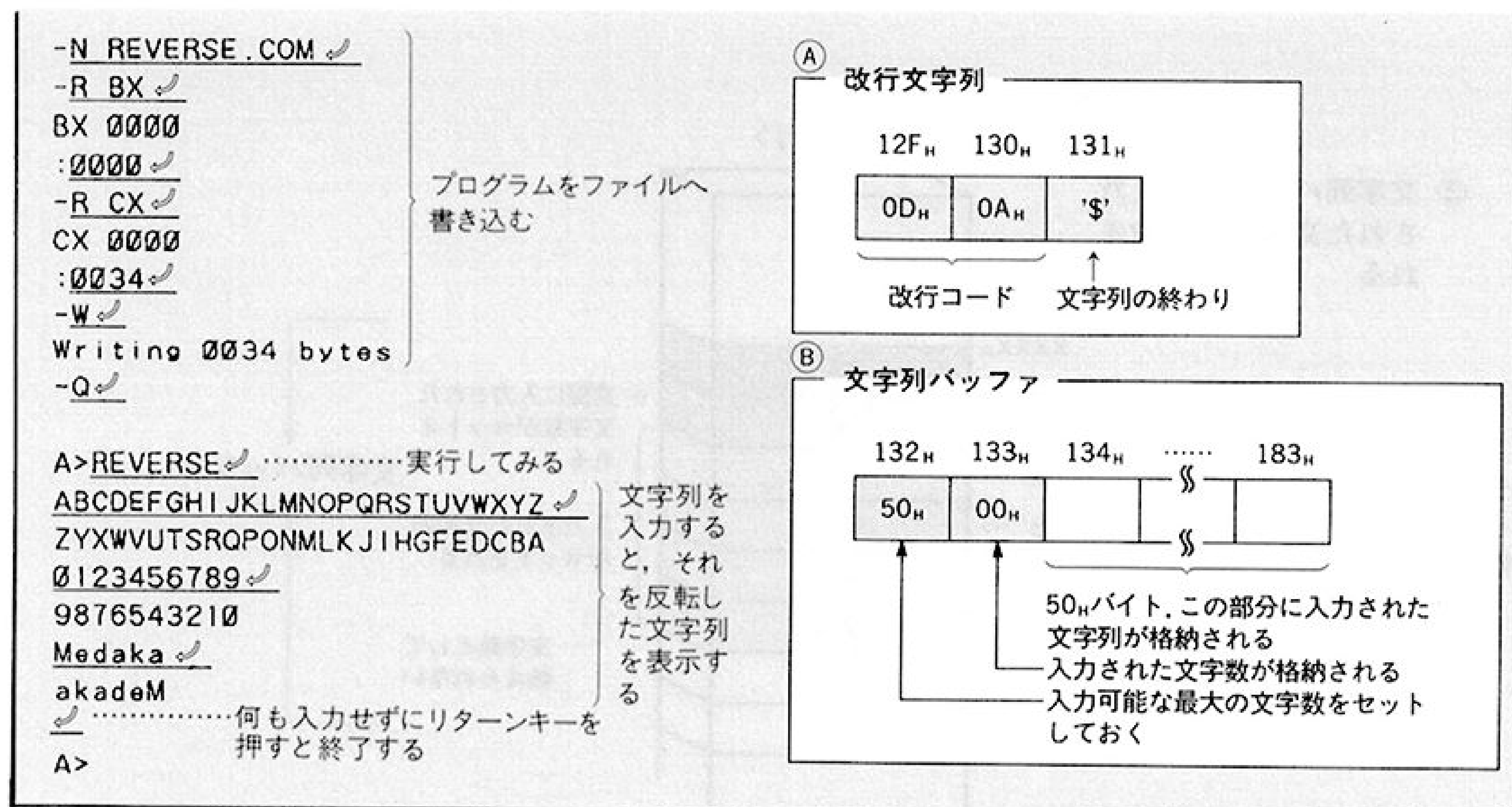


図 7-4　文字列反転プログラムと実行例

このプログラムのポイントは，BXレジスタを"文字列バッファを指し示すポインタ"兼"表示する文字数のカウンタ"として使用していることです．

まず，ファンクションコール$0A_H$番で文字列が入力されると，その文字数をBXレジスタにセットします．そして1文字表示するごとにBXレジスタをデクリメントし，0になるまで繰り返します．さらに，文字列バッファから表示する文字を1文字ずつ取り出すためのポインタとして，BXレジスタを使用しています．この仕組みを図 7-5 に図解しますので，BXレジスタがどこを指しているのかをよく理解してください．

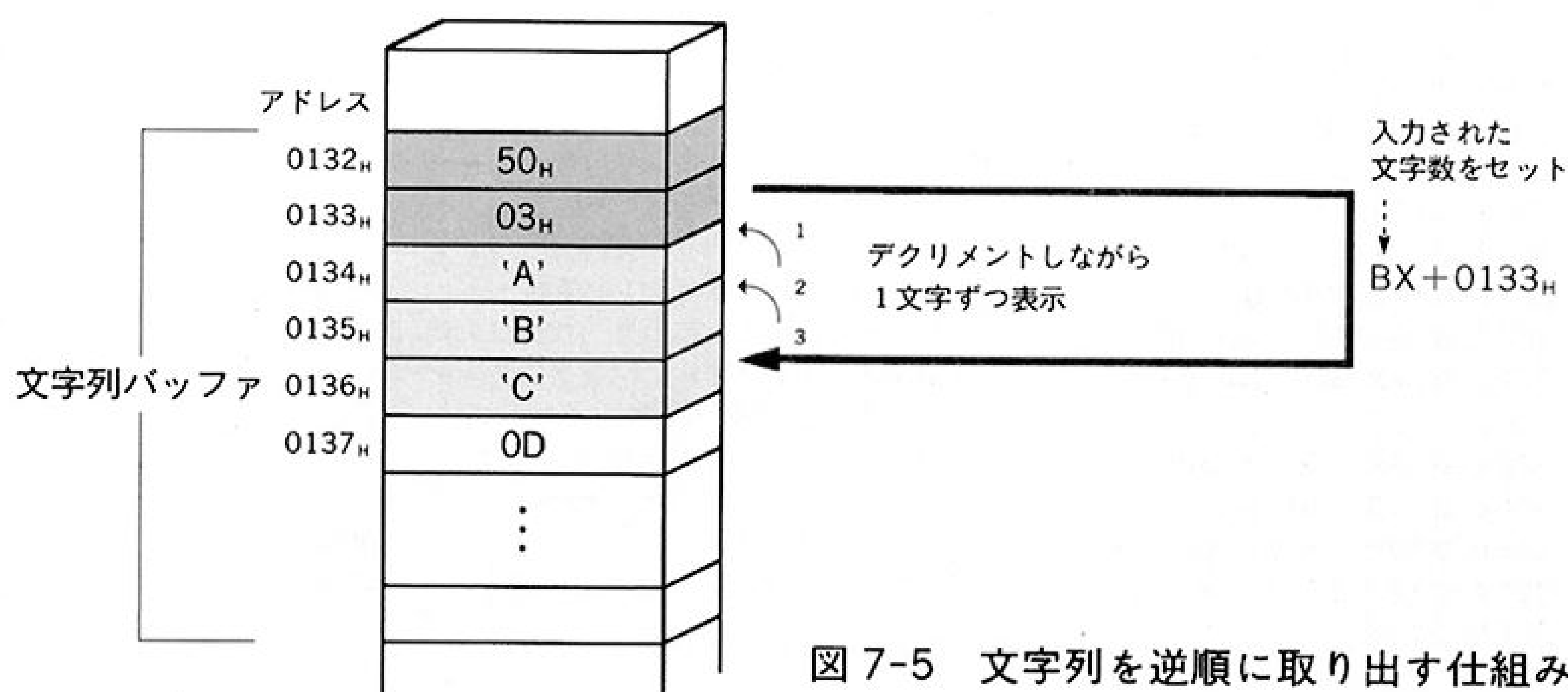


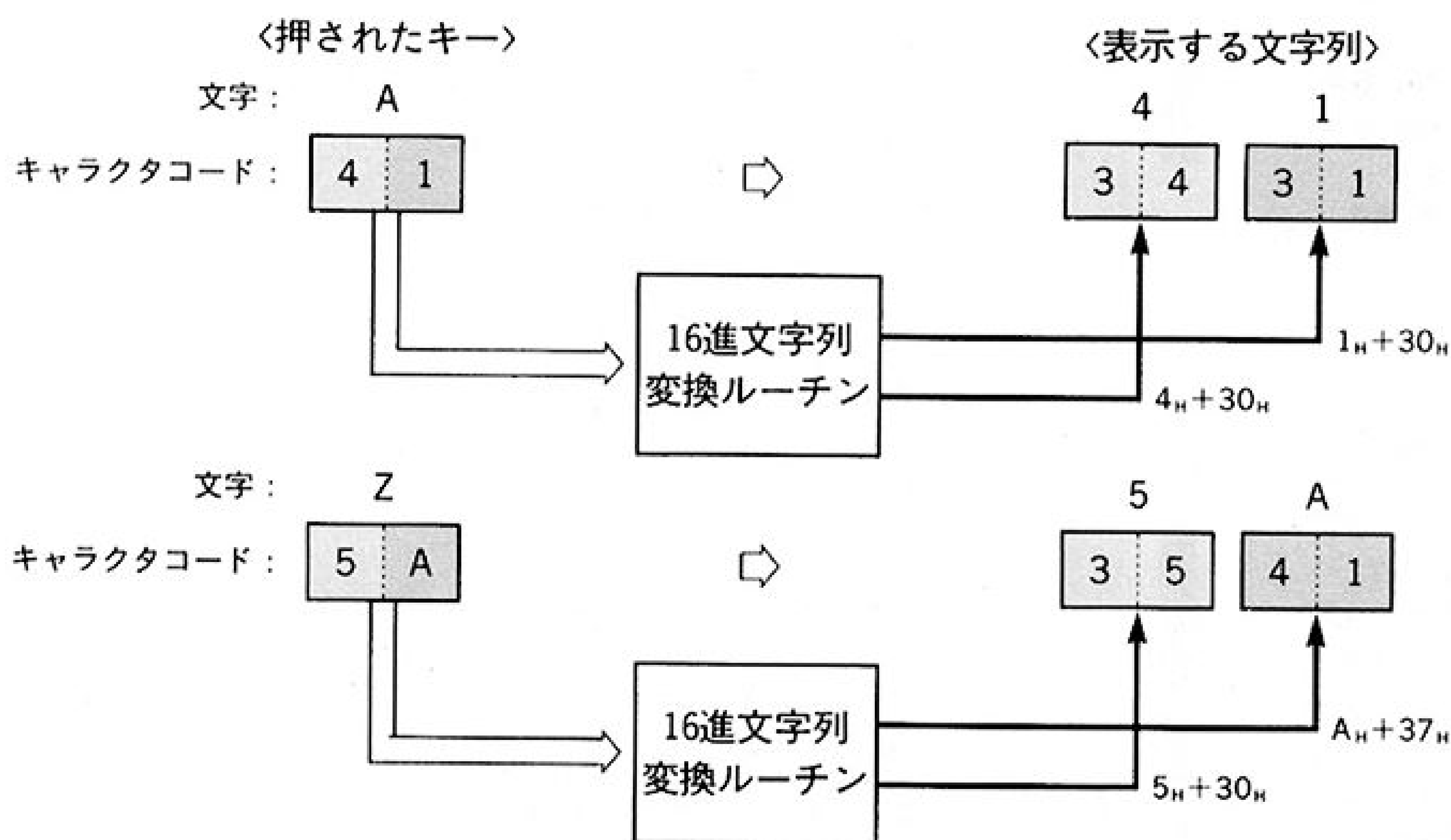
図 7-5　文字列を逆順に取り出す仕組み

7.3　キャラクタコード表示プログラム

最後はもう少し高級なものに挑戦しましょう．ここでは文字を入力し，その文字に対応するキャラクタコードを16進数で表示するプログラムを作成します．たとえば，「A」というキーを押すと，「A」とさらにそのキャラクタコードである16進数の「41」を表示します（273ページのキャラクタコード表を参照）．

キーボードのそれぞれのキーには，キャラクタコードそのものが割り当てられているので，それを表示するのは実に簡単に思えます．しかし，ここでちょっと考えてみてください．「A」という文字に対して「41」と表示するにはどうすればよいのでしょうか？

そのためには，文字のキャラクタコードを数字のキャラクタコードに変換しなければならないのです．次の図7-6を見てください．



（キャラクタコードの上位／下位4ビット）

0～9の場合 ⇨ 30_Hを加算（文字0のキャラクタコード30_Hと数字0_Hの差）
A～Fの場合 ⇨ 37_Hを加算（文字Aのキャラクタコード41_Hと数字A_Hの差）

図7-6　文字のキャラクタコードを数字のキャラクタコードに変換する

これがプログラムの核となる部分です．この変換部分は，サブルーチンとします．このプログラムの中では１ヶ所でしか呼び出しが行われずサブルーチンにする必要はとくにないのですが，それ自体で意味を持っており１つの処理単位になっています．このような場合はほかのプログラムで利用することもできますから，独立したサブルーチンにしておいたほうがよいでしょう．

なお，このプログラムで使用するファンクションコール９番では，DX レジスタの内容をオフセットアドレスとするメモリから連続するメモリの内容を文字列として画面に表示します．文字列の最後は「＄」で指定します．つまり，指定されたアドレスから「＄」のキャラクタコードを見つけるまで，アドレスをインクリメントしながらメモリの内容を文字として表示していくのです．

それでは実際に，プログラムを作成し実行した例を示しましょう（図 7-7）．

```
A>DEBUG↵
-A 0100↵
3CFB:0100 MOV AH,01       } コンソールから１文字入力（ファンクションコール１番）
3CFB:0102 INT 21          }
3CFB:0104 MOV DL,AL       ……入力した文字（ALレジスタの内容）をDLレジスタにセット
3CFB:0106 MOV DI,0133     ……文字列を格納するバッファの末尾のアドレスをDI レジスタにセット
3CFB:0109 CALL 0115       ……16進文字列変換サブルーチンをコール
3CFB:010C MOV AH,09       } DXレジスタにセットされたアドレスからの文字列を
3CFB:010E MOV DX,0132     } コンソールに表示（ファンクションコール９番）
3CFB:0111 INT 21          }
3CFB:0113 JMP 0100        ……オフセットアドレス0100H番地へジャンプ
                          [メインルーチン]
3CFB:0115 PUSHF           ……フラグをスタックへ退避（ディレクションフラグを変更するため）
3CFB:0116 STD             ……ディレクションフラグをセット
3CFB:0117 MOV BL,02       ……BLレジスタに桁数２をセット
3CFB:0119 MOV CL,04       ……CLレジスタにシフト数４をセット
3CFB:011B MOV AL,DL       } 入力した文字の下位４ビットをALレジスタにセット
3CFB:011D AND AL,0F       }
3CFB:011F CMP AL,09       } ALレジスタの内容が'9'より大きければ，
3CFB:0121 JG 0127         } オフセットアドレス0127H番地へジャンプ
3CFB:0123 ADD AL,30       ……'0'のキャラクタコード30HをALレジスタに加算
3CFB:0125 JMP 0129        ……オフセットアドレス0129H番地へジャンプ
3CFB:0127 ADD AL,37       ……（'A'のキャラクタコード41H）－AH＝37HをALレジスタに加算
                          （0117～0127 ⇨Ⓐ）
3CFB:0129 STOSB           ……ALレジスタの内容をDI レジスタの指すメモリにストアし，DI レジスタをデクリメント
3CFB:012A SHR DL,CL       ……入力された文字のビットを４ビット右にシフト
                            （上位４ビットが次の１桁であるため）
3CFB:012C DEC BL          } 文字数カウンタ（BLレジスタ）をデクリメントし，０でなければ
3CFB:012E JNZ 011B        } オフセットアドレス011BH番へジャンプ
3CFB:0130 POPF            ……フラグをスタックから復帰
3CFB:0131 RET             ……サブルーチンからメインルーチンへのリターン
                          [16進文字列変換サブルーチン]
3CFB:0132 DB '  H',0D,0A,'$'  ……画面に表示する文字列のセット ⇨Ⓑ
3CFB:0138
```

```
-N KEYCODE.COM↵
-R BX↵
BX 0000
:0000↵
-R CX↵
CX 0000
:0038↵
-W↵
Writing 0038 bytes
-Q↵

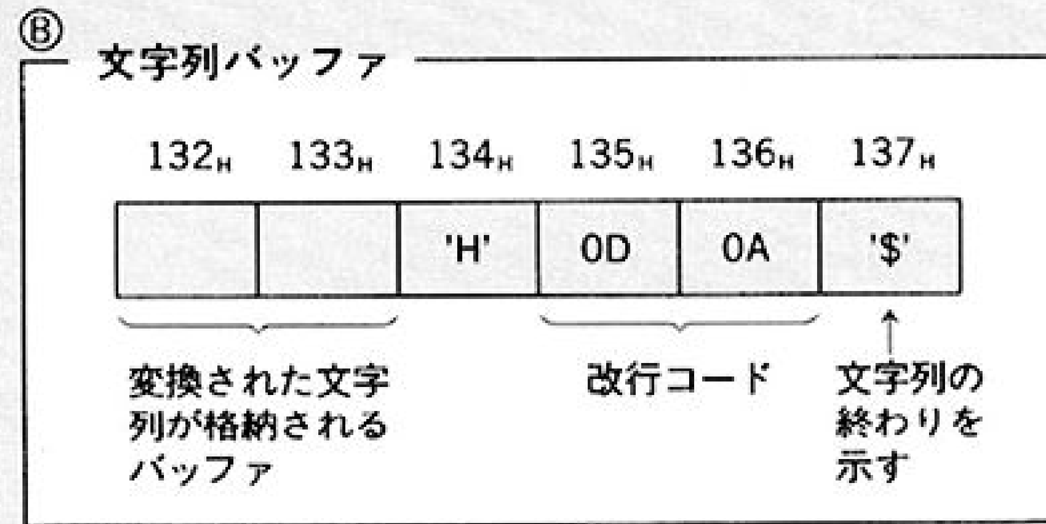
A>KEYCODE↵
A41H
a61H
131H
232H
$24H
^C
A>
```

プログラムをファイルへ書き込む（-N KEYCODE.COM～-W）

A>KEYCODE↵ ……実行してみる

キーを押すと，それに対応するキャラクタコードが16進数で表示される（A41H～$24H）

^C ……CTRL+C で終了する



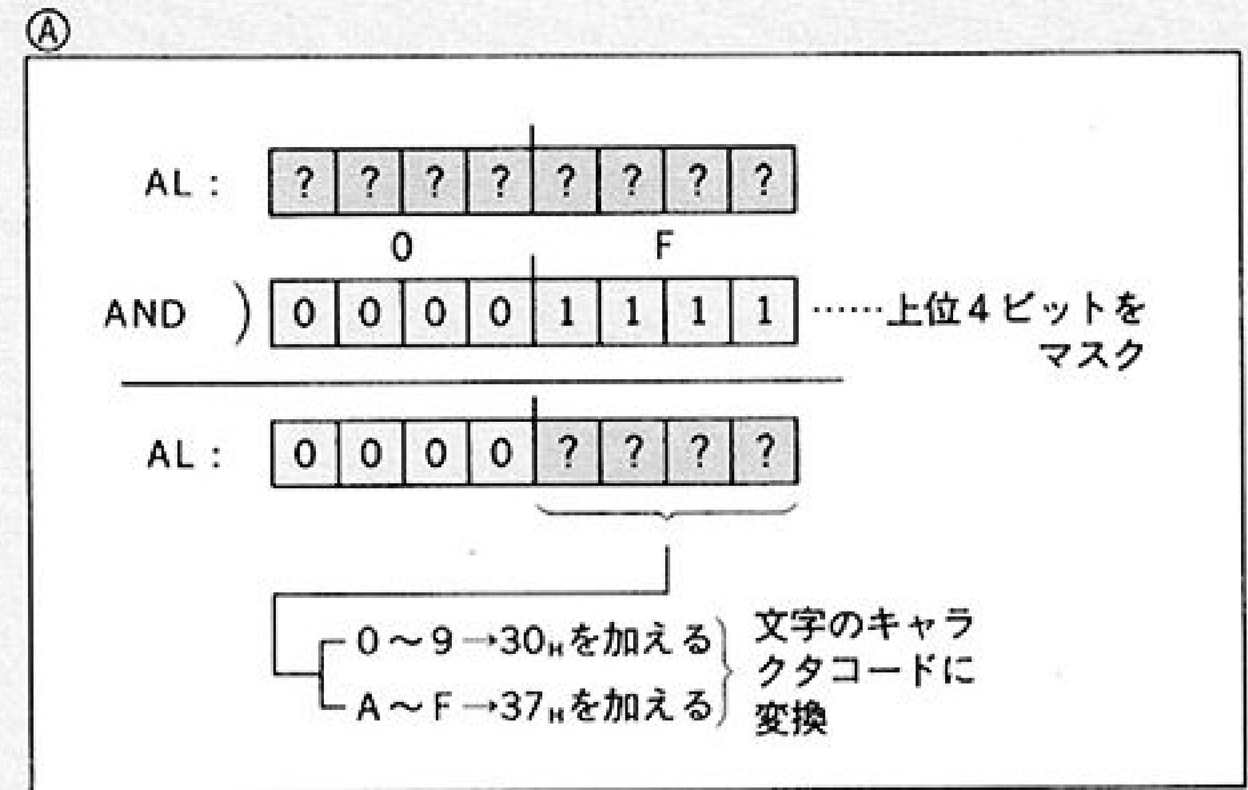
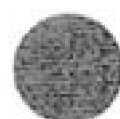


図 7-7 キャラクタコード表示プログラムと実行例

このプログラムの16進文字列変換サブルーチンでは，4ビットのデータを取り出すためのAND命令，上位4ビットを下位4ビットにシフトするためのSHR命令，そして変換されたデータをストアするためのSTOSB命令など各種の命令を使っていますから，図7-7のコメントを参照してじっくりと解析してみてください．

8

マクロアセンブラによる マシン語プログラミング



これまではマシン語の入力／実行ツールとしてDEBUGコマンドを使ってきました．DEBUGはマシン語プログラムを入力し，それをその場で確認することのできるたいへん便利なツールです．しかし，本格的なマシン語プログラムの作成となると，DEBUGでは手に負えません．

そこで本章では，実際のマシン語プログラミングで使われるマクロアセンブラというツールを紹介します．MS-DOSにはMASMと呼ばれるマクロアセンブラやLINKというリンカなど，いくつかのマシン語プログラム開発用のツールがあり，ここではそれらを使ってマシン語プログラムを作成してみることにしましょう．

この章ではページ数の関係もあり最低限の知識しか紹介できませんが，これまで学習したマシン語の知識と合わせて，より本格的なマシン語プログラミングへの足掛りとなることを期待します．

また最後には，2，3章で使ったCDUMPコマンドをMASMを使って作成することにします．

8.1　DEBUGとマクロアセンブラ（MASM）

●

ここでは，まずマクロアセンブラ（MASM）とはいったいどういうツールなのかという話から始めます．同じマシン語プログラミングでもDEBUGとマクロアセンブラでは，その手法が大きく異なっており新たな知識も必要になります．そこで，これまで実習で使ってきたDEBUGコマンドとマクロアセンブラを対比しながら，マクロアセンブラの持つ機能の概要やプログラミングの流れを見ていくことにしましょう．

●

DEBUG本来の機能とは…

DEBUGは「デバッガ」と呼ばれるように，本来はデバッグ（プログラムの虫をとる）のためのツール（コマンド）です．そのためにデバッグに役立つ多くの機能が用意されており，本書でもこれまで，

Aコマンド（アセンブル）	…	ニーモニックをマシン語に変換する
Uコマンド（逆アセンブル）	…	マシン語をニーモニックに変換する
Dコマンド（ダンプ）	…	メモリの内容を16進数でダンプする
Eコマンド（エンター）	…	メモリの内容を変更する
Gコマンド（ゴー）	…	マシン語プログラムを実行する
Tコマンド（トレース）	…	マシン語プログラムをトレースする

などのコマンドを使ってマシン語の実習を行ってきました．本書ではコンピュータの動作原理や8086CPUの機能を理解してもらうことを主眼に置いているので，最も簡単で，しかも豊富な機能を持った「DEBUG」コマンドを使ってきたのです．DEBUGの機能を図解すると次の図8-1のようになるでしょう．

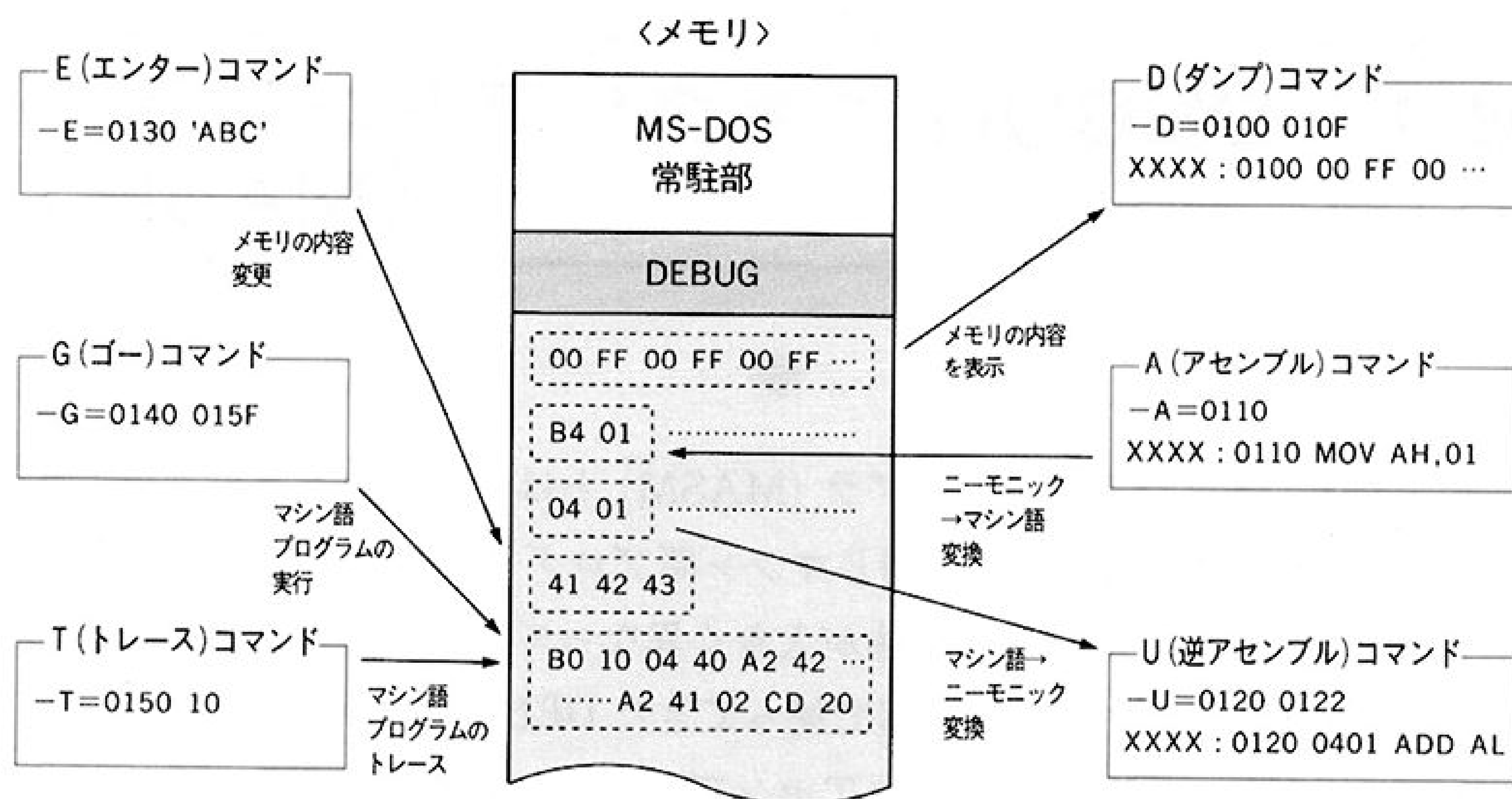


図 8-1　DEBUG コマンドの機能

しかし，DEBUGはあくまでプログラムのバグをとるためのデバッキングツールであり，本格的なマシン語プログラムの作成に使うことはできません（デバッキングツールとしてのDEBUGコマンドの使い方は,「8.3 MASMによるプログラム開発」で取り上げることにします）.

これまでもDEBUGを使って入力ミスをしたとき（リターンキーを押してしまった場合）の修正に不便を感じた人も多いでしょう．また，ジャンプ命令で飛び先のアドレス（現在のアドレスより先のアドレスにジャンプする場合）がどうしてわかるのか不思議に思っていた人もいると思います．

これらの不満や疑問を解決してくれるのが，これから紹介するMASMなのです．

●

マクロアセンブラ(MASM)の機能

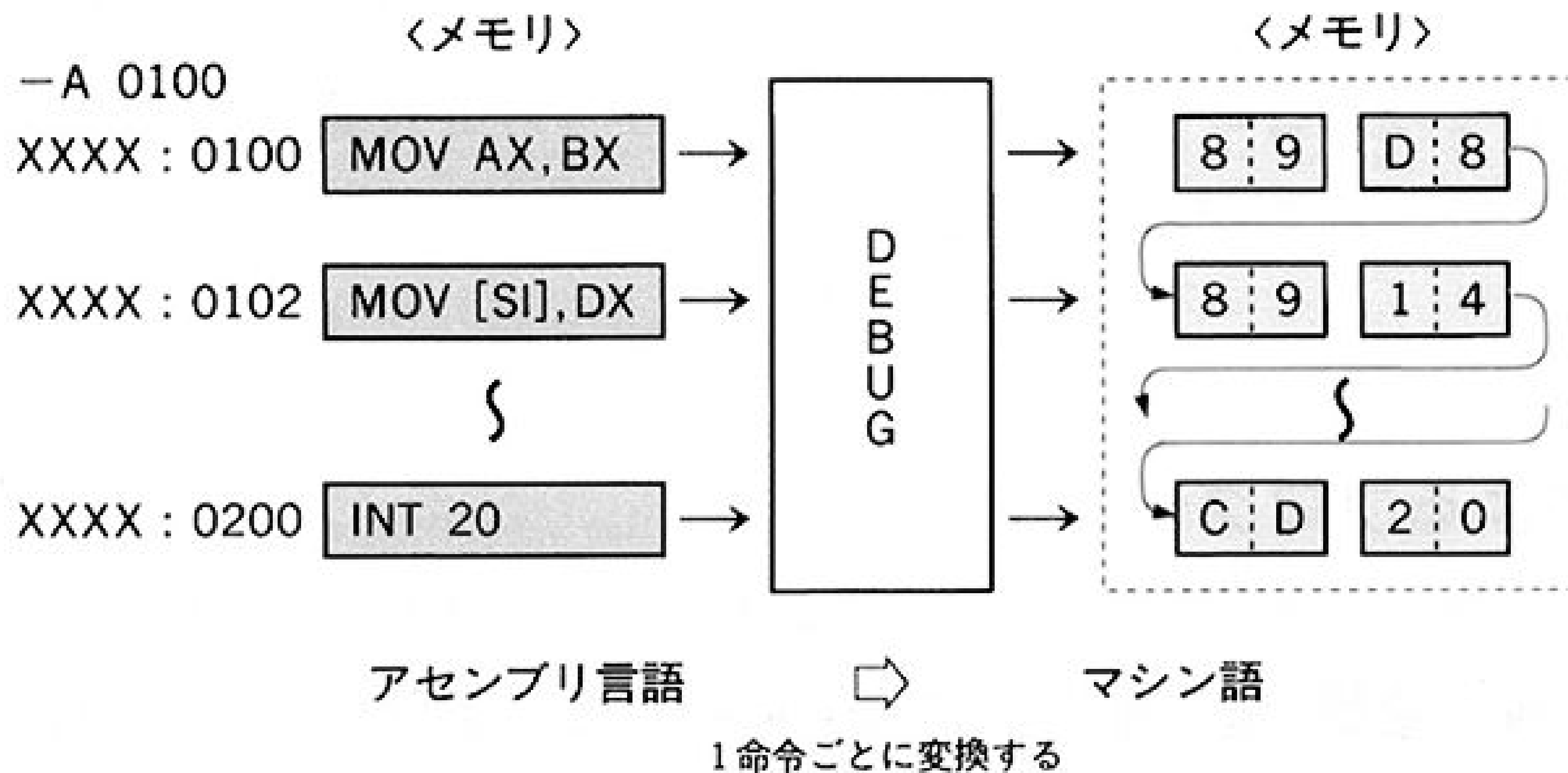
DEBUGのAコマンドのように，アセンブリ言語のニーモニックからマシン語に変換することを「アセンブル」すると言いますが，このアセンブルを専門に行うツールが「アセンブラ」です．MS-DOSには「MASM」というアセンブラが用意されています*.

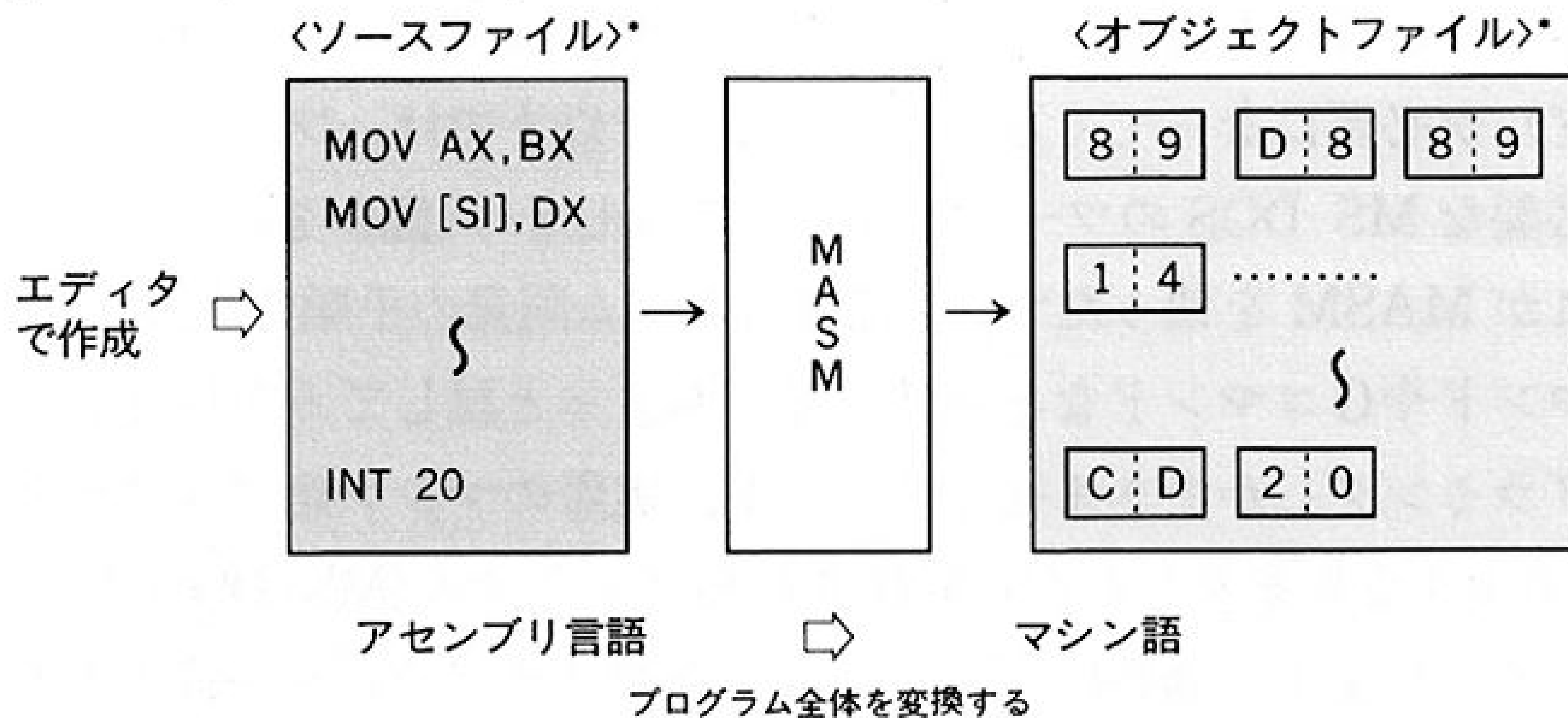
---

* MS-DOSのシステムディスクとは別売されている機種もある

MASMはマクロアセンブラとも呼ばれているように,単にアセンブリ言語をマシン語に変換する（アセンブルする）だけでなく，マシン語プログラムを開発するための強力な機能を持っています.本書ではMASMの優れた機能については紹介できませんが,MASMを使ってアセンブラの基本的な機能を押さえていくことにしましょう.

DEBUGでアセンブルする場合には，アセンブリ言語のニーモニックを1行入力するたびに，その命令がマシン語に変換されていました．これに対しアセンブラでは，アセンブリ言語で書かれたプログラム全体をマシン語に変換します．アセンブラを使う場合には，プログラムをテキストファイルとしてあらかじめ作成しておく必要があります.DEBUGによるアセンブルとMASMよるアセンブルの違いを図解すると次の図8-2のようになるでしょう.





＊ソースファイルとオブジェクトファイルについては，次項で解説する．

図8-2 DEBUGのAコマンドとMASMによるアセンブル

プログラムはエディタを使って作成します．本書ではエディタに関する解説は行いませんが，プログラムを作成するための必須のツールですから，ぜひ使えるようになってください．MS-DOS には「EDLIN」というエディタが標準で用意されています．EDLIN はラインエディタという形式で行単位でしか処理できないものですが，スクリーンエディタという使いやすい形式のものも市販されています．また，ワープロソフトでも MS-DOS の標準テキストファイル（TYPE コマンドで読めるファイル）を出力できるものならば，プログラムを作成するエディタとして利用することができます．

プログラムはマシン語の命令をアセンブリ言語のニーモニックで1行ずつ書き並べたものです．MASM でアセンブルするためにはいくつかの約束事を守らなければなりませんが，これについては次の 8.2 節で解説することにしましょう．このように作成したプログラムを「ソースプログラム」，プログラムを収めたテキストファイルを「ソースファイル」と呼びます．また，机上で紙などに書いてみることを含めて，ソースファイルを作成することを「コーディング」と呼ぶことがあります．

●

MASM によるプログラミング

DEBUG では，そのなかでアセンブルやダンプ，実行などのマシン語プログラムの作成や編集，実行までのすべてを行うことができますが，MASM ではプログラムのアセンブルしか行うことができません．つまり，MASM でマシン語プログラムを開発するためには，MASM だけでなく各種のツール（コマンド）が必要になってくるのです．そこで以下では，プログラム開発の手順と必要な MS-DOS のツールを示してみましょう（図8-3）．

これが MASM を使ったマシン語プログラム開発の手順です．DEBUG では A コマンドや G コマンドなどを使って〝ちょっと試してみる〟という程度のプログラミングしかできませんでしたが，実際のマシン語プログラミングでは，このような作業をしながら完成されたプログラム（MS-DOS のコマンド）を作っていきます．通常はミスのない完全なプログラムを一度できちんと作成することはむずかしく，必ずどこかでミスがあるものです．そのために，ソースファイルの作成から実行までの作業をデバッグしながら何度も繰り返

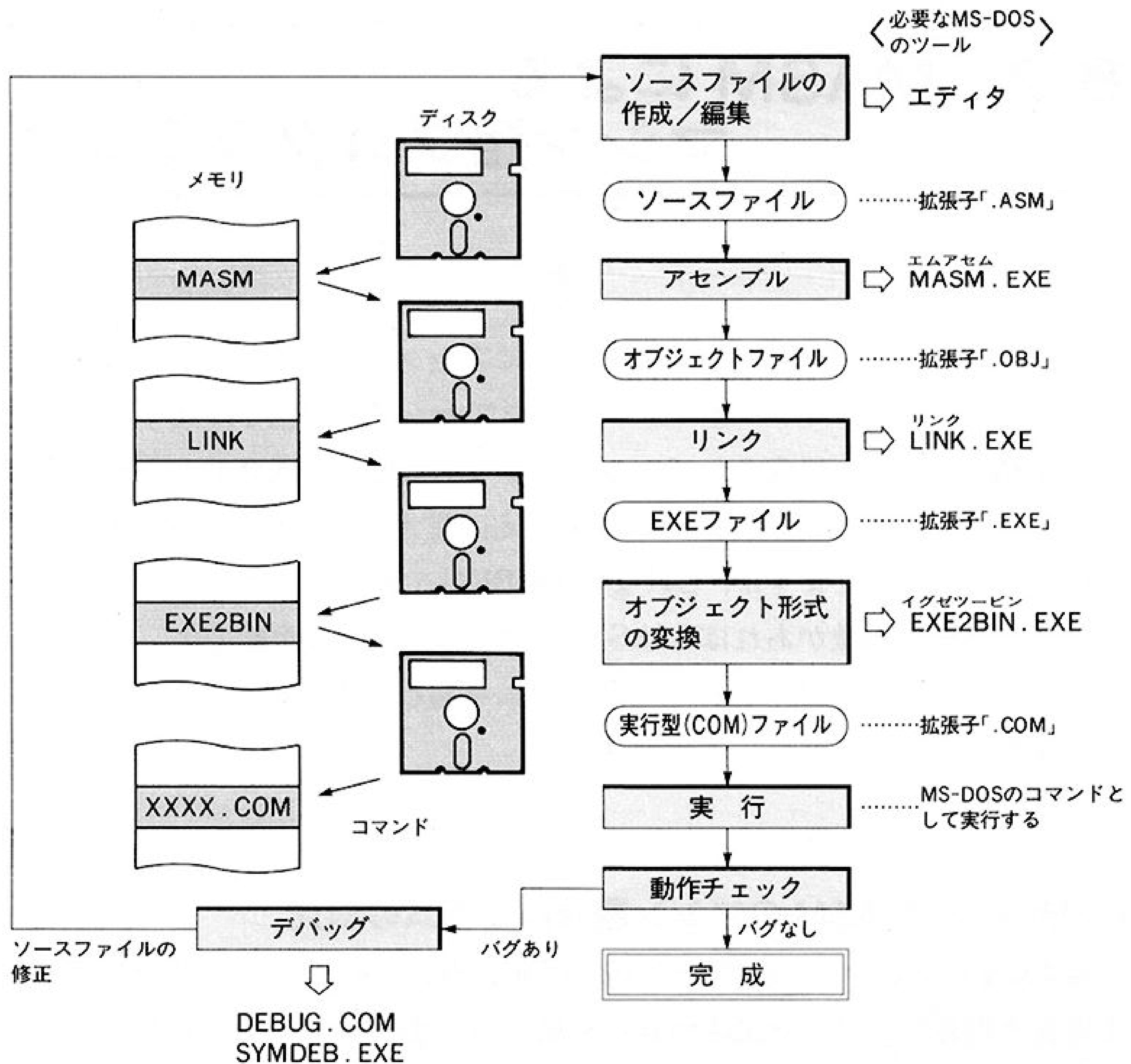


図 8-3　MASM によるプログラム開発の手順

すことになります.

また, 図 8-3 にあるように 1 つの作業ごとにファイルが作られることに注目してください. このようなファイルを作っておくことで, たとえば前に作成した同じようなソースファイルを利用したり, ある機能を持った完成したモジュール（プログラムの一部）をプログラムに組み込んだりというような効率的なプログラム開発ができるのです.

このようなプログラム開発の実際の方法については,「8.3 MASM によるプログラム開発の手順」で実行例をまじえながら取り上げることにしましょう. ここでは, MASM を使ったマシン語プログラムの開発はこのような方法で行うのだということを頭に入れておいてください.

8.2 MASMによる マシン語プログラミング

MASMは，これまでも述べてきたようにアセンブルを行うことしかできません．そのかわり，マシン語プログラムをアセンブリ言語で記述するための便利な機能が豊富に用意されているのです（本書では，残念ながらそのすべてを紹介することはできません）．逆にいえば覚えることが多く，むずかしいということにもなりますが，これまでDEBUGでやってきたマシン語のニーモニック＋αの知識があればMASMを使うことは十分可能です．

ここでは，MASMでマシン語プログラムを組むために最低限必要な決まり事を解説していくことにしましょう．

DEBUGとMASMのマシン語プログラムの対比

MASMによるマシン語プログラムをDEBUGのAコマンドでアセンブルする場合と対比して，その記述の違いを見ていきます．ここでは6章の実習10で取り上げた「入力されたキーがアルファベット大文字かどうかを判定するプログラム」を例とします．以下にDEBUGで記述した場合とMASMで記述した場合のプログラムを示します．

図8-4の中で網をかけた部分がDEBUGとMASMのプログラムで異なるところです．MASMのソースプログラムの方が前後に余分な命令が付き，プログラム中に特定のアドレスが出てこないことがわかるでしょう．この図では，プログラムを書くための表記法に関わる部分（■色の部分）と，プログラムには直接関係ないがアセンブラ（MASM）がアセンブルを行うために必要とする指示の部分（■色の部分）を色で区別しています．以降では，この2つを切り分けて解説していきます．

〈DEBUGのAコマンドでアセンブルする場合のマシン語プログラム〉

```
-A Ø1ØØ                   → オフセットアドレス0100Hからアセンブルする
3A9F:Ø1ØØ MOV AH,1
3A9F:Ø1Ø2 INT 21
3A9F:Ø1Ø4 CMP AL,41       ┐
3A9F:Ø1Ø6 JB Ø115         │ DEBUGでは数値は16進数として扱われる
3A9F:Ø1Ø8 CMP AL,5A       ┘
3A9F:Ø1ØA JA Ø115         ジャンプ先はオフセットアドレスで指定する
3A9F:Ø1ØC MOV AH,9
3A9F:Ø1ØE MOV DX,Ø11E
3A9F:Ø111 INT 21
3A9F:Ø113 JMP Ø1ØØ        データがストアされているオフセット
3A9F:Ø115 MOV AH,9        アドレスを指定する
3A9F:Ø117 MOV DX,Ø13D
3A9F:Ø11A INT 21
3A9F:Ø11C JMP Ø1ØØ
3A9F:Ø11E DB ' アルファベット大文字である',ØD,ØA,'$'
3A9F:Ø13D DB ' アルファベット大文字でない',ØD,ØA,'$'
3A9F:Ø15C
```

〈エディタで作成したMASMのソースプログラム〉

```
CODE    SEGMENT
        ASSUME CS:CODE,DS:CODE,ES:CODE,SS:CODE
        ORG Ø1ØØH
```
各セグメントレジスタがどのセグメントに割り当てられるか，またプログラムの先頭のオフセットアドレスをどこに設定するかをMASMに指示する

```
START: ……………(コード)ラベル
        MOV AH,1
        INT 21H ……………MASMでは数値は10進数として扱われるので，16進数を用いる場合は最後に「H」を付ける
        CMP AL,'A' ………シングルクォートで囲んだ文字はキャラクタコードに置き換えられる
        JB NOT_ALPHA
        CMP AL,'Z'     ジャンプ先の指定にオフセットアドレスではなくラベルが使用できる
        JA NOT_ALPHA
        MOV AH,9
        MOV DX,OFFSET ALPHA_MES
        INT 21H
        JMP START      ラベルに「OFFSET」を付けると，ラベルで
NOT_ALPHA: ……………(コード)ラベル   指定したメモリのオフセットアドレスを表す
        MOV AH,9
        MOV DX,OFFSET NOT_ALPHA_MES
(データ)ラベル INT 21H
        JMP START
ALPHA_MES      DB ' アルファベット大文字である',ØDH,ØAH,'$'
NOT_ALPHA_MES  DB ' アルファベット大文字でない',ØDH,ØAH,'$'

CODE    ENDS
        END START
```
プログラムの終了とプログラムの実行開始アドレスをMASMに指示する

図 8-4　DEBUG と MASM のマシン語プログラムの対比（6 章の実習 10 のプログラム）

マクロアセンブラの文法　ープログラムの表記法ー

まず，マシン語命令の書き方，つまりプログラムの表記法から見ていくことにしましょう．

【ラベル】

アセンブラを使用することによる最大のメリットは，「ラベル」が使えるという点です．ラベルとは特定のアドレスに付けた名前であり，アセンブラではアドレスを数値ではなく必ずラベルで表現します．

DEBUGを使った実習では，ジャンプ命令やコール命令の飛び先を直接オフセットアドレスで指定していました．この場合，現在のアドレスよりも先にジャンプする際には，そのアドレスを自分で計算しなければなりません*．しかし，MASMではこれをラベルで指定することで，そのアドレスがどんな値になるのかを知る必要がないのです．

たとえば上のプログラムでは，「START」と「NOT_ALPHA」というラベルが使われています．

```
START:
```

のように「：」（コロン）が付いているのがラベルです．ラベルはその次の命令が格納されているアドレスを意味します．このようにプログラムのコード（命令）部分のアドレスに付けたラベルを，後で解説するデータ部分に付けるラベルと区別して，「コードラベル」と呼ぶことにします．この場合はプログラムの先頭アドレスにSTARTという名前を付けたことになります．

```
NOT_ALPHA:
```

という行は，その次の「MOV　AH, 9」という命令が格納されるアドレスに名前を付けたことになります．そして，

```
JB   NOT_ALPHA
```

という命令では，MASMがソースプログラムをアセンブルする際に「NOT_

＊6章の実習では，ジャンプ先のアドレスはあらかじめ計算してあるものとして実習を行った．

ALPHA」を「MOV　AH, 9」の先頭アドレスに置き換えるのです．つまり，アセンブラはプログラムの先頭から命令の総バイト数を数えて，ラベルに対応するアドレスを自動的に決定してくれることになります．

ラベルには自由に好きな名前を付けることが可能です*．この例ではアルファベットでない場合の処理を「NOT_ALPHA」としましたが，これを「NOA」としても「ABC」としてもかまいません．といってもわかりやすい名前を付けることが肝心です．

ラベルを使うことで，プログラムの作成や修正がずいぶん楽になります．MASM ではソースプログラムをエディタで作成するので，DEBUG でアセンブルするのに比べて命令の挿入や削除が簡単に行えるのはもちろんですが，その際にプログラムの長さが変化しても，ジャンプ命令やコール命令の飛び先のアドレスを気にする必要はないのです．

【数値表現】

アセンブラではいろいろな数値表現を使うことができます．DEBUG では数値はすべて 16 進数であるとして扱われましたが**，アセンブラでは単に数値を書くと 10 進数であるとみなされます．16 進数を表すには数値の後に「H」を付けます．また，「'A'」のように「'」（シングルクォーテーション）で文字を囲んだものは，その文字のキャラクタコードに置き換えられます．

```
CMP   AL ,'A'   →   CMP   AL , 41H
```

また文字列を「'」で囲んだものは，そのキャラクタコードを並べたデータ列に置き換えられます．

【データを指し示すラベル】

MASM のプログラムでキーが押されたときに表示する文字列データには，「ALPHA_MES」と「NOT_ALPHA_MES」というラベルがそれぞれ付けられています．このようにプログラムのデータ部分に付けたラベルを「デー

＊正確に言うと，ラベルは「A～Z，0～9，?，@，_，$」からなる文字列で，「0～9」で始まることはできない．ラベルの文字数は任意であるが，31 文字を超える文字は無視される．また，アセンブラの予約語（ニーモニックや擬似命令など）と同じ文字列は使えない（使える場合もある）．

＊＊ SYMDEB では 2 進，8 進，10 進，16 進の数値を扱うことができる．

タラベル」と呼ぶことにします*.

データラベルはDB命令などのデータをセットする命令に対して付けることができます．また，コードラベルと違って「：」を付けないことに注意してください．

さて，データラベルの扱いは，DEBUGなどで直接アドレスを指定する場合と大きく異なる点があります．たとえば，

MOV　AL , [0400] …… アドレス0400_H番地のメモリの内容をALレジスタにロード

という命令を，データラベルを使って書くと，

MOV　AL , MEMORY …… データラベル「MEMORY」で指定するアドレスのメモリの内容をALレジスタにロード

となり，データラベルを指定するとそのアドレスのメモリの内容であると解釈されます．これに対して，データのアドレスそのものを参照したいという場合はどうするのでしょうか？　その場合は，「OFFSET」という演算子を使います．先の図8-4では，MS-DOSのファンクションコール9番で表示する文字列の先頭アドレスをDXレジスタにセットする際に，

MOV　DX , OFFSET ALPHA_MES …… データラベル「ALPHA_MES」で指定するアドレスをDXレジスタにロード

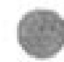
としてラベルの前に「OFFSET」という演算子を付けています．「ALPHA_MES」だけではそのアドレスのメモリの内容を意味することになりますが，OFFSET演算子を付けることでそのアドレスそのものを表すことができるのです．

コードラベルおよびデータラベルの表記法が，DEBUGでの表記とどう対応するかをまとめておきましょう．

---

*ここでいう「データラベル」とはマクロアセンブラの用語で「変数」と呼んでいるものにあたる．同様にコードラベルは単に「ラベル」と呼ばれる．本書ではわかりやすくするために「コードラベル」，「データラベル」とした．

MASM での表記		DEBUG での表記
コードラベル	←	アドレス
データラベル	←	[アドレス]
OFFSET データラベル	←	アドレス



マクロアセンブラの文法　－プログラム以外に必要な擬似命令－

マシン語のニーモニック以外の命令はマシン語に変換されるわけではなく，アセンブラに対する指令なので**擬似命令**と呼ばれています．先に挙げたラベルやOFFSET演算子，それにDB命令なども擬似命令です．

MASMでアセンブルを行うためには，プログラム以外にも書いておかねばならない擬似命令がいくつかあります．ここでは「COMモデル」（拡張子に「.COM」の付いた実行型ファイル）のプログラムを作成する場合に必要な擬似命令について説明します*．

【SEGMENT】

実行型ファイルのプログラムは，「5.5　セグメントの考え方」の図5-13（109ページ参照）で説明したようなセグメントの集まりという形式をとっています．そして，プログラムやデータは必ずいずれかのセグメントに属していなければなりません．「SEGMENT」擬似命令はセグメントを定義するための擬似命令であり，次のような形式で使用します．

名前　SEGMENT

プログラムやデータ

名前　ENDS

セグメントには任意の名前を付けることができます．プログラムやデータはSEGMENT擬似命令からENDS（END SEGMENT）擬似命令までの間に書くことによってそのセグメントに属することになります．

＊このほかに，「EXEモデル」（拡張子に「.EXE」の付いた実行型ファイル）があるが，本書では解説しない．

【ASSUME】

MASM がプログラムやデータのアドレスを決定するためには，セグメントレジスタがどのセグメントに割り当てられているかを知らなければなりません．それを MASM に指示するのが「ASSUME」擬似命令です．ASSUME 擬似命令は次のような形式で使用します．

ASSUME　セグメントレジスタ：セグメント名

セグメントレジスタは CS, DS, ES, SS のいずれかを指定し，セグメント名は SEGMENT 擬似命令で設定したセグメントの名前を指定します．「，」(カンマ) で区切ることで，いくつかのセグメントレジスタについて同じように指定することもできます．

【ORG】

MASM では特に指定しない限り，各セグメントのプログラムやデータはオフセットアドレス 0000_H から配置されます．しかし，COM モデルではプログラムの先頭のオフセットアドレスが 0100_H に固定されているので，スタートアドレスを設定しなければなりません．そのための擬似命令が「ORG」擬似命令です．

ORG 擬似命令は次のような形式で使用します．

ORG　オフセットアドレス値

COM モデルのプログラムの場合は常に，「ORG　0100H」とします．これは DEBUG でプログラムを入力するときに，「－A　0100」としてアセンブルするアドレスを指定していたことにあたります．

【END】

ソースファイルの終わりには「END」擬似命令が必要です．END 擬似命令はパラメータとしてプログラムの実行開始アドレス（コードラベル）をとります．したがって次のような形式で使用します．

END　プログラムの実行開始アドレス(コードラベル)

COM モデルのプログラムでは実行開始アドレスは必ず 0100_H なので，ORG 擬似命令の直後にコードラベルを定義してそのラベルを指定します．

以上の 4 つの擬似命令が COM モデルのプログラムのソースファイルに必要な擬似命令です．これをまとめるてみると，COM モデルのプログラムは次の図 8-5 のような形式で書くことになります．

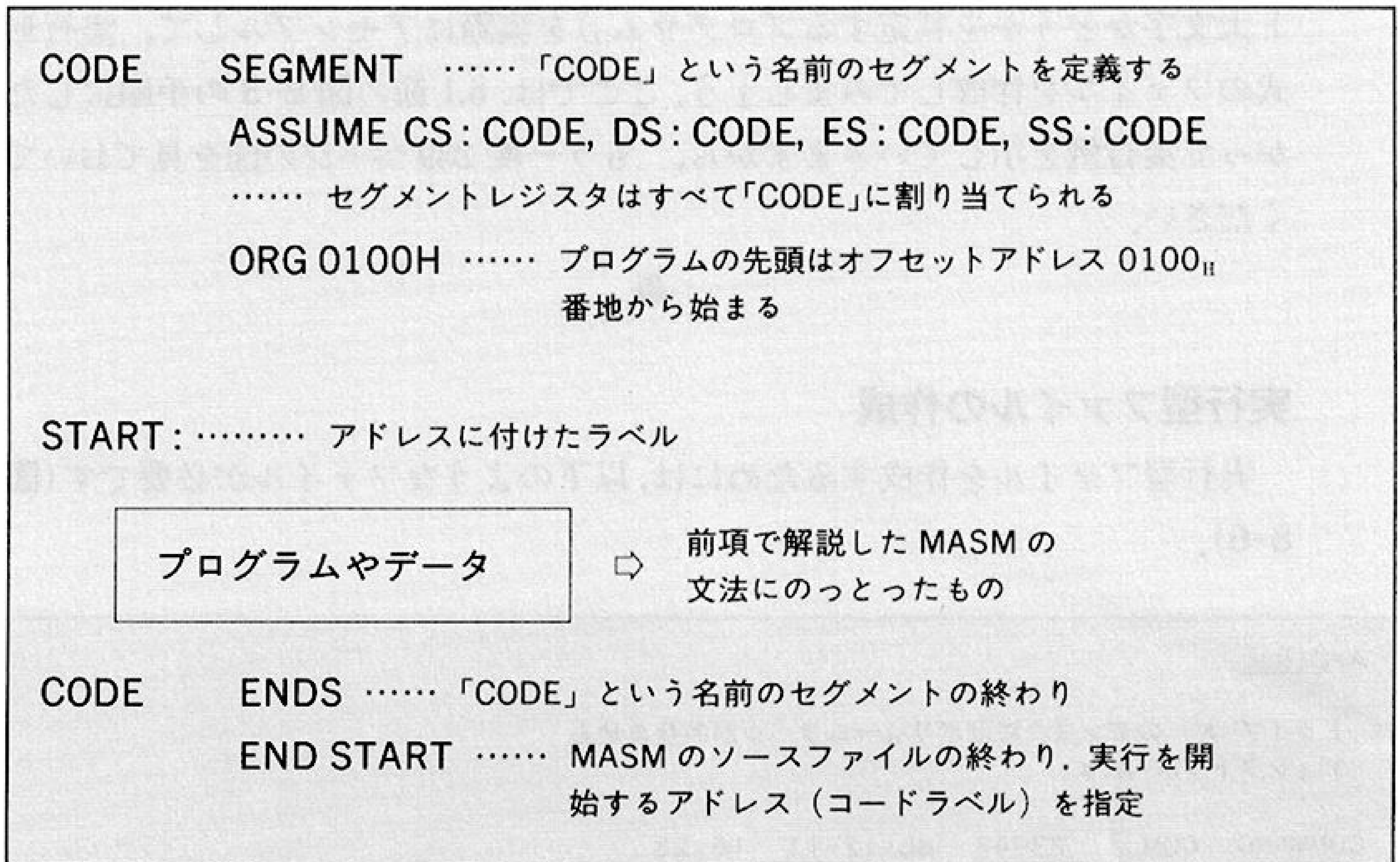


図 8-5　実行型ファイル（COM モデル）を作成するために必要な擬似命令

この図では，セグメント名を「CODE」，プログラムの実行開始（スタート）アドレスのコードラベルを「START」としました．これはほかの名前でもかまいません．図 8-5 の四角で囲んだ部分には，前項で解説した MASM の文法にそったプログラムを記述します．

これで，MSAM でアセンブル可能なソースファイルが完成しました．

8.3 MASMによる プログラム開発の手順

●

前節で作成したMASMのソースファイル(「入力されたキーがアルファベット大文字かどうかを判定するプログラム」)を実際にアセンブルして，実行形式のファイルを作成してみましょう．ここでは，8.1節の図8-3の手順にしたがって実行例を示していきますから，もう一度239ページの図を見ておいてください．

●

実行型ファイルの作成

実行型ファイルを作成するためには，以下のようなファイルが必要です(図8-6)．

```
A>DIR

 ドライブ A: のディスクにはボリュームラベルがありません
 ディレクトリは A:¥

COMMAND  COM    23942  85-12-11  15:28
MASM     EXE    77362  84-11-21  14:49 ……………マクロアセンブラ
LINK     EXE    41322  85-01-17  16:05 ……………リンカ
EXE2BIN  EXE     1656  83-09-04  21:14 ……………オブジェクト形式変換プログラム
DEBUG    COM    11764  83-02-01  10:13 ……………デバッガ
ALPHA    ASM      565  87-02-01  18:17 ……………前節で作成した「キー入力判定プログラム」
        6 個のファイルがあります                  のソースファイル
  1029120 バイトが使用可能です

A>
```

図8-6　実行型ファイルを作成するのに必要なファイル

図8-6で，「ALPHA.ASM」というファイルが前節で作成したプログラムのソースファイルになります．また，このほかプログラムを修正するためにエディタも入れておくとよいでしょう．

【アセンブル】

MASMでアセンブルするためのソースファイルには,「.ASM」というファイル拡張子を付けます．アセンブルは次の図8-7のように行います．

```
            ┌──→拡張子「.ASM」は省略できる
            │    ┌→セミコロンを付けると以後の指定が省略される
A>MASM ALPHA;↵ ……………「ALPHA.ASM」をアセンブルする
Microsoft MACRO Assembler  Version 3.00
(C)Copyright Microsoft Corp 1981, 1983, 1984


49698 Bytes free

Warning Severe
Errors  Errors   エラーがない．ここでエラーが表示されれば
0       0        ソースファイルに誤りがある

A>DIR ALPHA.*↵

 ドライブ A: のディスクにはボリュームラベルがありません
 ディレクトリは A:¥

ALPHA    ASM     565  87-02-01  18:17
ALPHA    OBJ     153  87-02-02  10:13 ……「ALPHA.OBJ」というオブジェクトファイル
         2 個のファイルがあります           が生成された
   1027072 バイトが使用可能です

A>

〈注意〉MASMはファイル名の指定などを対話形式で行うこともできるが，ここでは取り上げない
```

図8-7　アセンブルの実行

この図からわかるとおり「ALPHA.OBJ」というファイルが生成されています．これは「オブジェクトファイル」と呼ばれます．オブジェクトファイルの内容はDEBUGのAコマンドでソースプログラムを1行ずつアセンブルした結果のマシン語とほぼ同じですが，これを直接実行したり，TYPEコマンドで表示させることはできません．

また,MASMはプログラムがどのようなマシン語にアセンブルされたかを収めた「リスティングファイル」と呼ばれるファイルも合わせて出力させることができます．その場合は,

A> MASM　ALPHA,,ALPHA; ⏎

とすることで,「ALPHA.OBJ」と「ALPHA.LST」という2つのファイルが作成されます.リスティングファイルはテキストファイルですから,TYPEコマンドで表示させることができます(図8-8).

```
A>TYPE ALPHA.LST⏎ ··············リスティングファイルの内容を表示する
Microsoft MACRO Assembler  Version 3.00              Page    1-1      } ヘッダ
                                                             02-17-87
アドレス      マシン語          アセンブリ・ソースプログラム
 0000                        CODE    SEGMENT
                                     ASSUME CS:CODE,DS:CODE,ES:CODE,SS:CODE
 0100                                ORG 0100H

 0100                        START:
 0100   B4 01                        MOV AH,1
 0102   CD 21                        INT 21H
 0104   3C 41                        CMP AL,'A'
 0106   72 0D                        JB NOT_ALPHA
 0108   3C 5A      対応するマシン語  CMP AL,'Z'
 010A   77 09                        JA NOT_ALPHA
 010C   B4 09                        MOV AH,9
 010E   BA 011E R                    MOV DX,OFFSET ALPHA_MES
 0111   CD 21                        INT 21H
 0113   EB EB   アドレスはリンカに    JMP START
 0115           よって最終的に決  NOT_ALPHA:
 0115   B4 09   定される             MOV AH,9
 0117   BA 013D R                    MOV DX,OFFSET NOT_ALPHA_MES
 011A   CD 21                        INT 21H
 011C   EB E2                        JMP START
 011E   81 40 83 41 83 8B     ALPHA_MES       DB 'アルファベット大文字であ
        83 74 83 40 83 78     る',0DH,0AH,'$'
        83 62 83 67 91 E5
        95 B6 8E 9A 82 C5
        82 A0 82 E9 0D 0A
        24
 013D   81 40 83 41 83 8B     NOT_ALPHA_MES   DB 'アルファベット大文字でな
        83 74 83 40 83 78     い',0DH,0AH,'$'
        83 62 83 67 91 E5
        95 B6 8E 9A 82 C5
        82 C8 82 A2 0D 0A
        24

 015C                        CODE    ENDS
                                     END START
 ...
NOT_ALPHA_MES. . . . . . . . . . . . . . .
START. . . . . . . . . . . . . . . . . . .   L NEAR  0100   CODE

49698 Bytes free

Warning Severe
Errors  Errors
0       0

A>
```

凡例: □:命令　■:パラメータ　▒:データ

図8-8　リスティングファイルの内容

MASMでアセンブルを行うと，その際にアセンブルエラーが表示されるかもしれません．その場合はエラーメッセージをよく見て，原因を調べてください．エラーの発生した箇所はリスティングファイルを作成することによって調べます．最初のうちは「,」（カンマ）と「.」（ピリオド）を間違えるといったミスを犯しやすいものです．原因がわかったらソースファイルを修正して再度アセンブルします．エラーがなくなるまで次のステップには進めません．

【リンク】

アセンブルの次には「リンク」という作業を行います．リンクを行うプログラムが「リンカ」です．MS-DOSには標準のリンカとして「LINK.EXE」が用意されています．LINKはファイル拡張子が「.OBJ」のオブジェクトファイルを，拡張子が「.EXE」の実行ファイルに変換します（正確には後で解説するように，オブジェクトファイルを結合して実行ファイルを生成します）．

先にできた「ALPHA.OBJ」をリンクしてみます（図8-9）．

```
                    ┌──→ 拡張子「.OBJ」は省略できる
                    │ ┌→ セミコロンを付けると以後の指定が省略される
A>LINK ALPHA;⏎ ……………「ALPHA.OBJ」をリンクする

Microsoft 8086 Object Linker
Version 3.01 (C) Copyright Microsoft Corp 1983, 1984, 1985

Warning: no stack segment ……………警告が表示されるが，これは無視してかまわない

A>DIR ALPHA.*⏎

 ドライブ A: のディスクにはボリュームラベルがありません
 ディレクトリは A:¥

ALPHA    ASM      565  87-02-01  18:17
ALPHA    OBJ      153  87-02-02  10:15
ALPHA    LST     1891  87-02-02  10:15
ALPHA    EXE      860  87-02-02  10:20 ……「ALPHA.EXE」というEXEファイルが作成された
        4 個のファイルがあります            （ただし，このファイルはそのままでは実行できない）
  1018880 バイトが使用可能です

A>
     〈注意〉LINKはファイル名の指定などを対話形式で行うこともできるが，ここでは取り上げない
```

図8-9　リンクの実行

【オブジェクト形式の変換】

COM モデルとして作成したプログラムは，拡張子が「.EXE」のままでは実行できません．そこで，実行可能な COM ファイルを生成するに「EXE2BIN」コマンドを使って「EXE ファイル」を「COM ファイル」に変換します（図 8-10）．

```
                ┌──────────── 拡張子「.EXE」は省略できる
                │        ┌─── 拡張子「.COM」は必ず指定する
A>EXE2BIN ALPHA ALPHA.COM  ……………EXEファイルをCOMファイルに変換する

A>DIR ALPHA.*

 ドライブ A: のディスクにはボリュームラベルがありません
 ディレクトリは A:¥

ALPHA    ASM      565  87-Ø2-Ø1   18:17
ALPHA    OBJ      153  87-Ø2-Ø2   1Ø:15
ALPHA    LST     1891  87-Ø2-Ø2   1Ø:15
ALPHA    EXE      86Ø  87-Ø2-Ø2   1Ø:2Ø
ALPHA    COM       92  87-Ø2-Ø2   1Ø:22……実行可能なCOM形式のファイルが作成された
        5 個のファイルがあります
  1Ø16832 バイトが使用可能です

A>
```

図 8-10 COM ファイルの作成

【実行】

これで実行可能な MS-DOS のコマンドができあがりました．実行して動作を確認してみましょう（図 8-11）．

```
A>ALPHA ……………作成されたプログラムを実行する
1  アルファベット大文字でない
2  アルファベット大文字でない
A  アルファベット大文字である
B  アルファベット大文字である
x  アルファベット大文字でない
y  アルファベット大文字でない
^C ……………[CTRL]+[C] で実行終了

A>
```

図 8-11 実行結果を確認する

【デバッグ】

図8-11ではうまく動作したようですが，通常の場合，一度で期待どおりプログラムが動くことはまれです．その場合には，プログラムのデバッグを行ってその原因をつきとめなければなりません．

実はこれまでの実習で活躍してくれた「DEBUG」は，その名のとおりデバッグのための強力なツールです．DEBUGの各種のコマンドを使ってプログラムの動作をガッチリ監視し，効果的にバグの原因を探し出すことができます．

デバッグの様子を実行例で示すことはなかなかできませんが，さわりだけでも紹介しましょう．次の図8-12はDEBUGを起動し，デバッグを開始した部分の実行例です．

```
A>DEBUG ALPHA.COM↵ ……………DEBUGを起動して，「ALPHA.COM」をメモリ上に読み込む
-R↵ ……レジスタの内容を表示                  ┌─── 読み込んだプログラムのサイズ
AX=0000  BX=0000  CX=005C  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=30F8  ES=30F8  SS=30F8  CS=30F8  IP=0100   NV UP DI PL NZ NA PO NC
30F8:0100 B401          MOV     AH,01          └─ プログラムのスタートアドレス
-U 0100↵
30F8:0100 B401          MOV     AH,01
30F8:0102 CD21          INT     21
30F8:0104 3C41          CMP     AL,41
30F8:0106 720D          JB      0115
30F8:0108 3C5A          CMP     AL,5A
30F8:010A 7709          JA      0115
30F8:010C B409          MOV     AH,09
30F8:010E BA1E01        MOV     DX,011E
30F8:0111 CD21          INT     21
30F8:0113 EBEB          JMP     0100
30F8:0115 B409          MOV     AH,09          実際には文字列データだが，逆アセ
30F8:0117 BA3D01        MOV     DX,013D      ┌─ンブルすると無理やりニーモニックに
30F8:011A CD21          INT     21           │ 変換されてしまう
30F8:011C EBE2          JMP     0100         │
30F8:011E 8140834183    ADD     WORD PTR [BX+SI-7D],8341
-G=0100 0104↵ ……………とりあえず，0104H番地まで実行してみる
A ……………[A]のキーを押した
AX=0141  BX=0000  CX=005C  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=30F8  ES=30F8  SS=30F8  CS=30F8  IP=0104   NV UP DI PL [NZ] NA [PO] NC
30F8:0104 3C41          CMP     AL,41
-T ……………トレースして1命令ずつ実行する

AX=0141  BX=0000  CX=005C  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=30F8  ES=30F8  SS=30F8  CS=30F8  IP=0106   NV UP DI PL [ZR] NA [PE] NC
30F8:0106 720D          JB      0115                       比較の結果    パリティフラグ
-T                                                         ゼロフラグが  も変化している
                                                           1になった
```

図8-12 デバッグの実行例

モジュールの概念

これまでの実行例を通じて，なぜこのようにめんどうな作業が必要なのか疑問に思った人もいるでしょう．これらの作業は，実はプログラムの「モジュール化」という概念に密接に結びついています．そこで，以下では「モジュール」という概念について簡単に説明します．

ある程度大きなプログラムを作成する際には，１つのソースファイルですべてを記述するのではなく，いくつかの機能に分割しそれぞれを別のソースファイルとして作成するという方法が取られます．たとえば，あるプログラムは「ファイルを扱う処理単位」，「画面を扱う処理単位」，「データを管理する処理単位」などと分割することができるかもしれません．

こうして分類した各処理単位のことを「モジュール」と呼びます．プログラムがモジュールの集まりとして作られる様子を図 8-13 に示します．

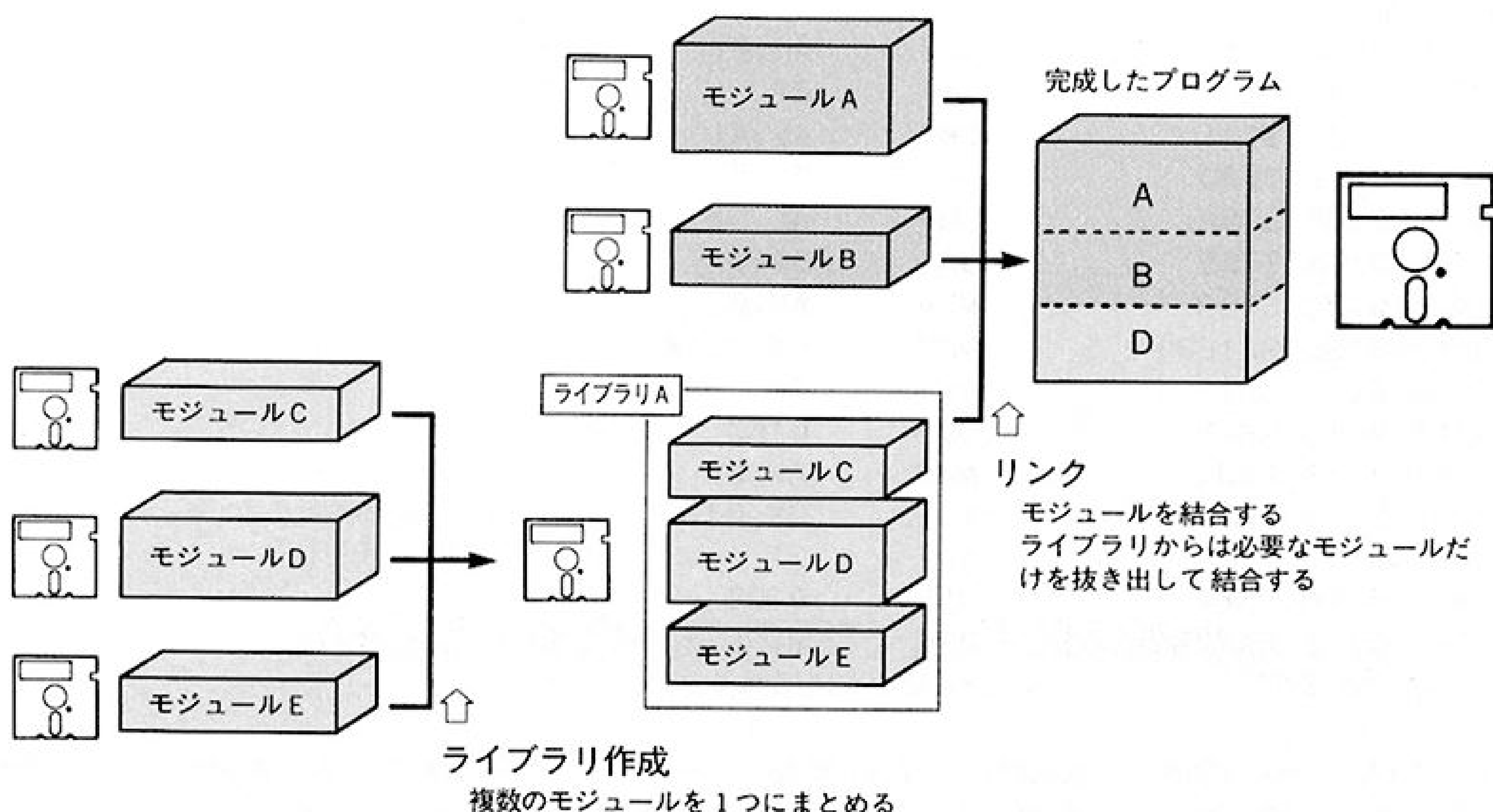


図 8-13　モジュールとプログラムの構造

この図のように，複数のモジュールを扱いやすいように１つにまとめておく場合があります．これを「ライブラリ」と呼び，リンク時にはライブラリから必要なモジュールを抜き出してリンクすることができます．

このようにプログラムをモジュールに分割しておけば，テストプログラムを作ることによって単体で動作確認をすることもでき，プログラムの仕様に変更があった場合でも関係のあるモジュールを変更するだけですみます．つまり，モジュールの集合体としてプログラムを作成することによって，プログラムの信頼性が高まり生産性が向上するのです．

MASMで作成するマシン語プログラムでもモジュールに分割することができます．MASMでは各モジュール単位にソースファイルを作成し，それぞれ別々にアセンブルを行います*．そして，最終的に完成したプログラムを作成するためにモジュールをまとめあげる作業が「リンク」なのです．次の図8-14にアセンブルからリンクまでのモジュール別プログラム開発の流れを図解しておきましょう．

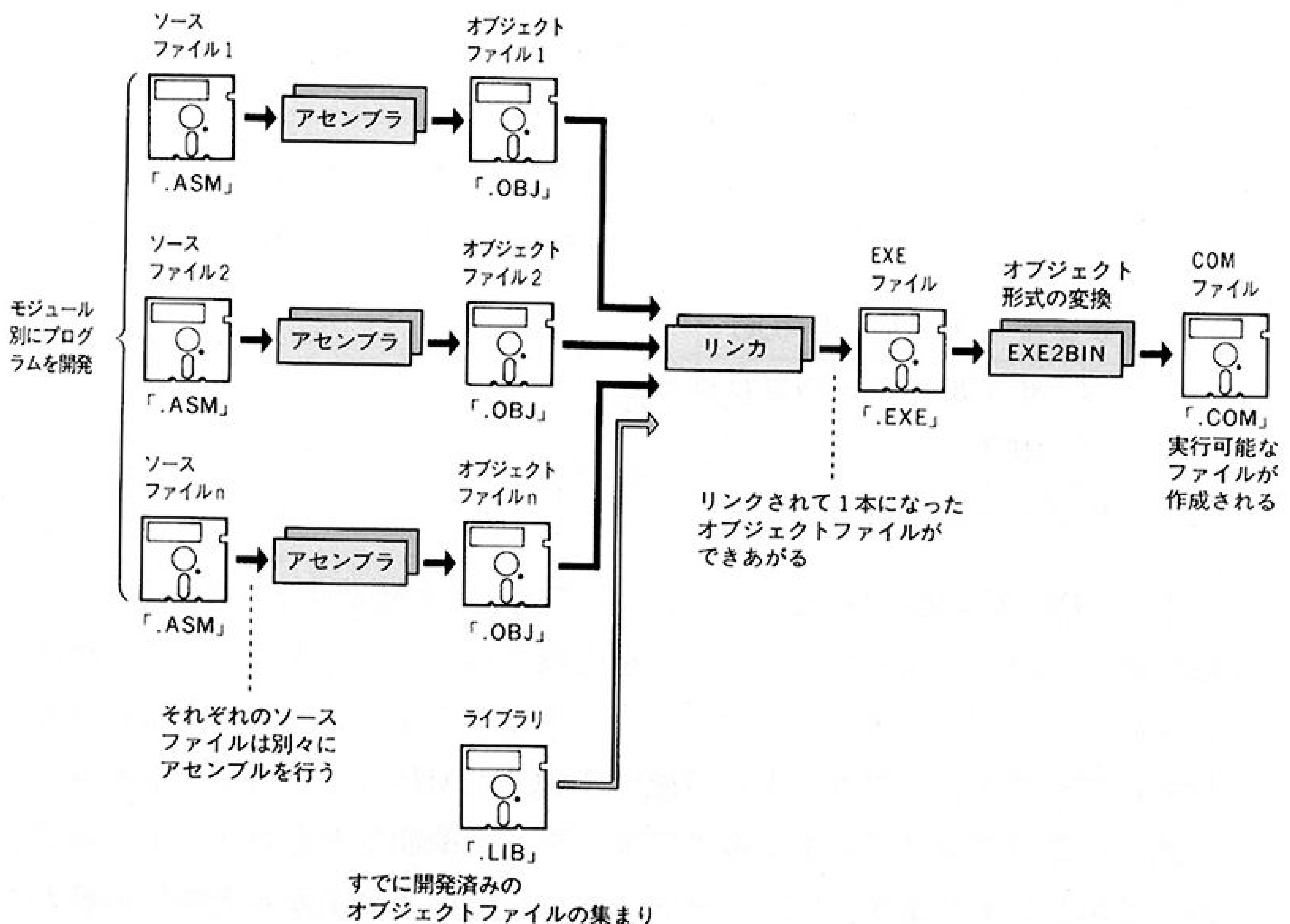


図8-14　モジュール別プログラム開発の流れ

＊各モジュール間での変数やサブルーチンの呼び出し方，ライブラリの利用法については，本書では取り上げない．くわしくは「応用MS-DOS」（アスキー発行）などの書籍を参考にするとよい．

8.4 プログラミングの実例 ―CDUMPプログラム―

本章のまとめとして，MASMによるプログラミングの実習を行います．例題として取り上げるのは2,3章でたびたび使用したCDUMPプログラムです．

このプログラムは，これまでの実習で作成してきたプログラムに比べてかなり長いので，そのすべてを理解するのはむずかしいかもしれません．ここではとりあえず，リストのコメントを参照して概要だけでもわかってもらえれば結構です．そして，実際にアセンブルやリンクを行って実行形式のファイルを作ってみてください．

またMASMの実習として，7章で取り上げたマシン語プログラムなども，各自でMASM用に変更してみるとよいでしょう．

このプログラムでは，8.2節で紹介した擬似命令のほかに新たな擬似命令を使っています．それはサブルーチンを定義する「PROC」擬似命令です．

```
サブルーチン名　PROC
  ┌──────────────────┐
  │ サブルーチンのプログラム │
  │ RET              │
  └──────────────────┘
サブルーチン名　ENDP
```

この「PROC」擬似命令によって，セグメントを定義する「SEGMENT」擬似命令と同じようにサブルーチンを定義することができます．この擬似命令を使ってサブルーチンをブロックとして独立させることで，構造の理解しやすいプログラムを作ることが可能です（CDUMPのソースリストを参照）．

また，このプログラムでは各サブルーチンの機能などをコメント（注釈）として示してあります．「；」（セミコロン）という文字からその行の終わりまではコメントとみなされ，アセンブラはその部分を読み飛ばしてしまいます．1行の途中からコメント付けることもできます．

それでは以下にCDUMPプログラムのフローチャートとソースファイル，

そして実行形式のファイルを作成するまでの手順を示します．

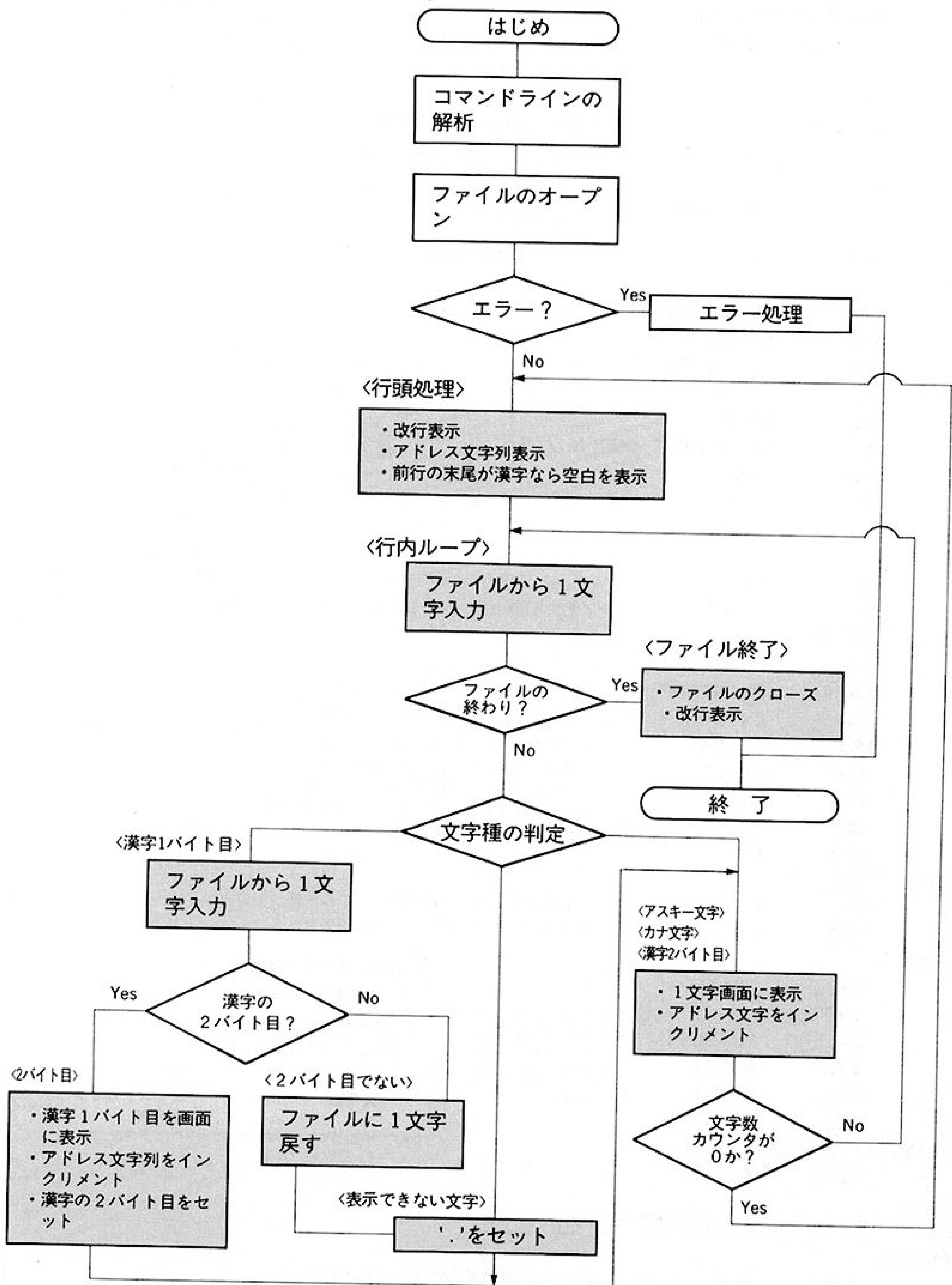


図8-15　CDUMPプログラムのフローチャート（メインルーチン）

```
;-----------------------------------------------
;           漢字ダンププログラム                      「;」(セミコロン)以降はコメント
;                  CDUMP.ASM                        何のプログラムかわかるようにタイ
;-----------------------------------------------    トルを付けておく
CSEG    SEGMENT                                      セグメントの設定
        ASSUME CS:CSEG,DS:CSEG,ES:CSEG,SS:CSEG
                               コマンドライン文字列が格納され
        ORG     8ØH            るエリア.これはアドレスをラベル  ⇒Ⓐ
CMDLN   DB      128 DUP (?)    で与えるためのダミーであり,実際
                               はMS-DOSがセットする

        ORG     1ØØH……………COMモデルのプログラムは
START:                     必ず0100Hから始まる
        MOV     BL,CMDLN   コマンドライン文字列の長さを
        XOR     BH,BH      BXレジスタに入れる
                                                            (ASCIZ文字列)
        MOV     CMDLN[BX+1],Ø……指定された文字列をファイル名とするために末尾に0を付加
        MOV     AH,3DH               ファイルをオープンするファンクションコール3DH
        MOV     AL,Ø                 AH : 3DH
        MOV     DX,OFFSET CMDLN + 2  AL : モード(0:読み出し,1:書き込み,
        INT     21H                        2:読み出し/書き込み)
                                     DX : ファイル名(ASCIZ文字列)の先頭アドレス
        JNC     OSUCC      キャリーフラグが1ならばファイルのオープンに失敗
        JMP     OERROR     「JC OERROR」としないのはショートジャンプではとどかないため
OSUCC:
        MOV     BX,AX ……オープンされたときのファイルの番号(ファイルハンドル)をBXレジスタにセット
        MOV     CX,Ø………CXレジスタを文字数カウンタとして使うため,0に初期化
LOOPL: [行頭処理]
        CALL    PRINTCRLF……改行サブルーチンをコール
        CALL    PRINTADR………アドレス部の表示サブルーチンをコール
        CMP     CX,Ø………………行数カウンタが0ならば前行の末尾が漢字でない
        MOV     CX,64……………文字数カウンタに1行に表示する文字数(半角)をセット
        JGE     LOOPC……………「LOOPC」へジャンプ(MOV命令ではフラグが変化しないので,
                                このジャンプは2行前のCMP命令の結果による)
        MOV     AL,' '     文字数カウンタが0でなければ,前の行の末尾が漢字である
        CALL    PUTC       前の行の最後で1バイト余分に出力しているので,1文字目は空白とする
        DEC     CX
LOOPC: [行内ループ]
        CALL    GETC……………ファイルから1バイト取り出すサブルーチンをコール
        JNZ     EOF ……………ゼロフラグが0ならば「EOF」へジャンプ
CHECK:
        CALL    ISKANJI    読み込んだ1バイトのデ
        JNZ     KANJI      ータが漢字コードの1バ
        CALL    ISASCII    イト目/アスキー文字/
        JNZ     ASCII      カナ文字かを調べ,それ
        CALL    ISKANA     ぞれの処理に分岐
        JNZ     KANA
NOTASCII: [表示できない文字]
        MOV     AL,'.'
ASCII:
KANA:  [アスキー文字,カナ文字,漢字コードの2バイト目]
KANJI2:
        CALL    PUTC…………………1文字表示するサブルーチンをコール
        CALL    INCADR……………アドレス文字をインクリメントするサブルーチンをコール
```

Ⓐ **コマンドラインの構造**

A>CDUMP TEST

05		'T'	'E'	'S'	'T'	0D

```
        DEC     CX……………………文字数カウンタをデクリメント          ｝「LOOP LOOPC」としない
        JG      LOOPC………………0より大きければ1行内ループを繰り返す｝のはCXが－1になること
        JMP     SHORT LOOPL…行頭処理へジャンプ                         がああるため

KANJI: 漢字コードの1バイト目
        MOV     KBUF,AL………読み込んだ文字をKBUFに退避
        CALL    GETC……………ファイルから1バイト取り出すサブルーチンをコールし，次の文字を読み込む
        JNZ     NOTASCII……ファイルの終わりならば，文字ではなかった
        CALL    ISKANJI2 ｝漢字の2バイト目かどうかを調べる．漢字の2バイト目
        JZ      NOTKANJI ｝でなければ表示できない文字である
        MOV     AH,KBUF  ｝KBUFとALレジスタの内容を交換する
        MOV     KBUF,AL  ｝  KBUF：漢字2バイト目，ALレジスタ：漢字1バイト目となる
        MOV     AL,AH    ｝     ⇨「XCHG AL, KBUF」と1命令で書ける
        CALL    PUTC……………1文字表示するサブルーチンをコールし，漢字1バイト目を表示
        CALL    INCADR…………アドレス文字列をインクリメントするサブルーチンをコール
        MOV     AL,KBUF………漢字2バイト目をALレジスタにロード
        DEC     CX…………………文字数カウンタをデクリメント
        JMP     SHORT KANJI2……………漢字の2バイト目の表示へ
NOTKANJI: 2バイト目でない
        CALL    UNGETC         ｝
        JMP     SHORT NOTASCII ｝余計に読み込んだ1バイトを戻す

EOF: ファイル終了
        MOV     AX,3EH ｝ファイルをクローズするファンクションコール3EH
        INT     21H    ｝ (AX：3EH
                          (BX：ファイルハンドル…オープンしたときのファイルハンドルがそのまま残っている
        CALL    PRINTCRLF………改行表示サブルーチンをコール
        JMP     RETURN……………プログラムの終了へ

;-------------------------------------------------------  ｝
;       文字がアスキー文字かどうかを調べる                   ｝
;         入力                                               ｝
;       AL  ：  文字                                         ｝サブルーチンの役割と
;         出力                                               ｝パラメータを記述して
;       Zフラグ ： セット      アスキー文字ではない          ｝おくとよい
;               ： リセット    アスキー文字である            ｝
;-------------------------------------------------------  ｝
ISASCII PROC
        CMP     AL,20H ｝
        JB      FALSE  ｝20H≦AL≦7EHならばアスキー文字である
        CMP     AL,7EH ｝(APPENDIXのキャラクタコード表参照)
        JA      FALSE  ｝
TRUE:
        OR      AH,0FFH………必ずゼロフラグがリセットされる        ｝
        RET                            ↓                          ｝この処理は以降の
FALSE:                  APPENDIXのフラグの変化を参照             ｝サブルーチンでも
                                       ↑                          ｝利用している
        CMP     AL,AL……………必ずゼロフラグがセットされる         ｝
        RET
ISASCII ENDP
```

```
;------------------------------------------------
;        文字が半角カナ文字かどうかを調べる
;          入力
;        AL  :  文字
;          出力
;        Zフラグ  :  セット      半角仮名文字ではない
;                 :  リセット    半角カナ文字である
;------------------------------------------------
ISKANA  PROC
        CMP     AL,0A0H          ┐
        JB      FALSE            │
        CMP     AL,0DFH          ├ A0H≦AL≦DFHならばカナ文字
        JBE     TRUE             │
        JMP     SHORT FALSE      ┘
ISKANA  ENDP

;------------------------------------------------
;        文字が漢字の1バイト目かどうかを調べる
;          入力
;        AL  :  文字
;          出力
;        Zフラグ  :  セット      漢字の1バイト目ではない
;                 :  リセット    漢字の1バイト目である
;------------------------------------------------
ISKANJI PROC
        CMP     AL,80H           ┐
        JBE     FALSE            │
        CMP     AL,9FH           │
        JBE     TRUE             │ 80H＜AL≦9FH ┐
        CMP     AL,0E0H          │    または    ├ ならば漢字コードの1バイト目
        JB      FALSE            │ E0H≦AL≦FCH ┘
        CMP     AL,0FCH          │
        JBE     TRUE             │
        JMP     SHORT FALSE      ┘
ISKANJI ENDP

;------------------------------------------------
;        文字が漢字の2バイト目かどうかを調べる
;          入力
;        AL  :  文字
;          出力
;        Zフラグ  :  セット      漢字の2バイト目ではない
;                 :  リセット    漢字の2バイト目である
;------------------------------------------------
ISKANJI2 PROC
        CMP     AL,40H           ┐
        JB      FALSE            │
        CMP     AL,7EH           │ 漢字コードの1バイト目に続く文字が
        JBE     TRUE             │   40H≦AL≦7FH
        CMP     AL,080H          │      または
        JB      FALSE            │   80H≦AL≦FCH
        CMP     AL,0FCH          │ ならば漢字コードの2バイト目
```

```
        JBE     TRUE
        JMP     SHORT FALSE
ISKANJI2 ENDP

;--------------------------------------------------
;       アドレス文字列を表示する
;--------------------------------------------------
PRINTADR PROC
        MOV     AH,9                   コンソールに文字列を表示するファンクションコール09H
        MOV     DX,OFFSET ADDRESS      (AH : 09H
        INT     21H                     DX : 文字列の先頭のオフセットアドレス
        RET
PRINTADR ENDP

;--------------------------------------------------
;       アドレス文字列を増加させる
;--------------------------------------------------
INCADR  PROC
        MOV     SI,OFFSET ADDRESS+5 ……………SIレジスタに末尾の桁のアドレスをロード
NEXTDIGIT:
        INC     BYTE PTR [SI]……………SIレジスタの指す桁の文字をインクリメント
        CMP     BYTE PTR [SI],'9'+1 } '9'より大きくなければリターン
        JL      ADREND              }
        JE      TOHEX ……………'9'+1ならば'A'にする
        CMP     BYTE PTR [SI],'F'   } 'F'より大きくなければリターン
        JLE     ADREND              }
        MOV     AL,'Ø'              } 'F'より大きければ,SIレジスタの指す桁を
        MOV     [SI],AL             } 0にして,SIレジスタをデクリメント
        DEC     SI                  }
        JMP     SHORT NEXTDIGIT ……………1つ大きい桁を同様に処理
TOHEX:
        MOV     AL,'A'              } '9'→'A'の処理
        MOV     [SI],AL             }
ADREND:
        RET
INCADR  ENDP

;--------------------------------------------------
;       改行する
;--------------------------------------------------
PRINTCRLF PROC
        MOV     AL,ØDH              } CRコードを表示(カーソルが行頭へ)
        CALL    PUTC                }
        MOV     AL,ØAH              } LFコードを表示(カーソルが1行下へ)
        CALL    PUTC                }
        RET
PRINTCRLF ENDP
```

```
;------------------------------------------------
;         入力バッファに1文字返す
;           入力
;         AL : 戻す文字
;------------------------------------------------
UNGETC  PROC
        MOV     BYTE PTR UNGETF,1    漢字処理のため，余分に読んだデータをとっておく
        MOV     UNGETBUF,AL          UNGETFを1にセットし，余分なデータがあることを示す
        RET
UNGETC  ENDP

;------------------------------------------------
;         ファイルから1文字読み込む
;           入力
;         BX : ファイルハンドル
;           出力
;         AL : 入力された文字
;         Zフラグ : セット     1文字読み込まれた
;                 : リセット   ファイルの終わり
;------------------------------------------------
GETC    PROC
        MOV     AL,UNGETF      漢字処理のために余分にデータを読んでいる場合は，
        CMP     AL,1           UNGETFが1にセットされている
        JNZ     RGETC
        MOV     BYTE PTR UNGETF,Ø……UNGETFを0にリセット
        MOV     AL,UNGETBUF …………………余分に読み込んであったデータを返す
        RET
RGETC:
        PUSH    CX ……………CXレジスタ（文字数カウンタに使われている）を退避
        MOV     AH,3FH          ファイルからデータを読み込むファンクションコール3FH
        MOV     DX,OFFSET BUF    (AH : 3FH
        MOV     CX,1             |BX : ファイルハンドル
        INT     21H              |CX : 読み込むバイト数
                                 (DX : データを読み込むオフセットアドレス
        JC      RERROR …………キャリーフラグが1ならエラー
        CMP     AX,CX ……………AXに読み込んだバイト数が返る，CXと比較し等しくなければファイル終了
        MOV     AL,BUF …………ALレジスタに読み込んだデータをロードする      ↓
        POP     CX ……………………退避しておいたCXレジスタを復帰   メインルーチンでジャンプ
        RET                                                    (以後の3命令ではフ
GETC    ENDP                                                    ラグは変化しない)

;------------------------------------------------
;         コンソールに1文字出力する
;           入力
;         AL : 出力する文字
;------------------------------------------------
PUTC    PROC
        MOV     AH,2
        MOV     DL,AL     コンソールに1文字表示するファンクションコール02H
        INT     21H        (AH : 02H
        RET                (DL : 表示する文字
PUTC    ENDP
```

```
エラー処理ルーチン
OERROR:
        MOV     DX,OFFSET OERRMES  } ファイルをオープンできなかったときの処理
        JMP     SHORT MSGOUT       }
RERROR:
        MOV     DX,OFFSET RERRMES ……………ファイル読み込みでエラーが発生したときの処理
MSGOUT:
        MOV     AH,9  } コンソールに文字列を表示するファンクションコール09H
        INT     21H   }  (AH : 09H
                         (DX : 文字列の先頭のオフセットアドレス
RETURN:
        INT     20H ……………プログラム終了のシステムコール
; ワークエリア
BUF     DB      0 ………ファイルからデータを読み込むためのバッファ
KBUF    DB      0 ………漢字コードを退避するバッファ
UNGETF  DB      0 ………漢字コードの処理のために，1バイト余分に読み込んだかどうかを示す変数
UNGETBUF DB     0 ………漢字コードの処理のために，1バイト余分に読み込んだデータを格納するバッファ
ADDRESS DB      '000000:   $' ……………アドレス表示用文字列
OERRMES DB      0DH,0AH,'Can't open',0DH,0AH,'$' …ファイルをオープンできないとき
RERRMES DB      0DH,0AH,'Read error',0DH,0AH,'$' … に表示するメッセージ
;                                            データ読み込みエラー時に表示するメッセージ
CSEG    ENDS                                 文字列中に「'」を含む場合は，「"」で囲む
;
        END     START
```

図 8-16　CDUMP プログラムのソースファイル

```
A>DIR

 ドライブ A: のディスクにはボリュームラベルがありません
 ディレクトリは A:¥

COMMAND  COM    23942  85-12-11   15:28
MASM     EXE    77362  84-11-21   14:49
LINK     EXE    41322  85-01-17   16:05
EXE2BIN  EXE     1656  83-09-04   21:14
CDUMP    ASM     4604  87-01-08   21:00 ……………CDUMPプログラムのソースファイル
         5 個のファイルがあります
   1037312 バイトが使用可能です

A>MASM CDUMP; ……………アセンブルを行う
Microsoft MACRO Assembler  Version 3.00
(C)Copyright Microsoft Corp 1981, 1983, 1984


48612 Bytes free

Warning Severe
Errors  Errors
0       0
```

```
A>LINK CDUMP;↵ ……………リンクを行う

Microsoft 8086 Object Linker
Version 3.01 (C) Copyright Microsoft Corp 1983, 1984, 1985

Warning: no stack segment

A>EXE2BIN CDUMP CDUMP.COM↵ ……………実行形式のファイル(COMファイル)に変換する

A>DIR CDUMP.*↵

 ドライブ A: のディスクにはボリュームラベルがありません
 ディレクトリは A:¥

CDUMP    ASM      4604  87-01-08  21:00
CDUMP    OBJ       653  87-02-02  17:17
CDUMP    EXE      1115  87-02-02  17:17
CDUMP    COM       347  87-02-02  17:18 ……………できあがった実行ファイル
         4 個のファイルがあります
   1030144 バイトが使用可能です

A>CDUMP B:DISKCOPY.COM↵ ……………CDUMPコマンドを実行してみる

000000:  颱 ......./.V.........................DISKCOPY version 2.1...D O
000041:   Sのバージョンが違います...送り側ディスクをドライブ @: に挿入し
000080:  てください...受け側ディスクをドライブ @: に挿入してください..準備
0000C1:   ができたらどれかのキーを押してください  .....ディスクのコピーを行
000101:   います.......ディスクの照合を行います.......コピーは終了しました
000141:   .....照合を終了しました...別のディスクをコピーしますか <Y/N>? .
000180:  ..別のディスクを照合しますか <Y/N>? .....コピーは失敗しました...
0001C0:  ..照合は失敗しました.....送り側ディスク.....受け側ディスク...中止
                                        中止 <A> ,強行 <I> ? .....パラメ
001641:   ..ｴ;ﾍ!s.鬢.ｺ..ｴ;ﾍ!s.髻.....罷
001680:  !s.馴.ﾆ....鞦.ｺr.ｴ.ﾍ!ﾆ....<.u.ﾆ....縀.ｺ..ｴ.ﾍ!<.u..>...u.魑.餔..>
0016C0:  ...u...ｾ..ｿ".ｹ...ｦt.  θ .ι .ｹ...ｦt.   .>...t.  占 .....閻.雍.ｺﾊ.ｴ
001700:  ;ﾍ!s.鰛.ｺﾙ.ｴ;ﾍ!s.鰆.....鎔.ｺr.ｴ.ﾍ!<.u.鰐.ｺr.ｴ.ﾍ!<.u.鮑..=.ｺ..ｴ.ﾍ
001740:  !<.u.魃.ｺ..ｴ.ﾍ!<.u.贇.闘..>...u.鬆....鍜.ｺﾊ.2ﾀｴ=ﾍ!s.騙.｣..ｺﾙ.2ﾀｴ
001780:  =ﾍ!s.駮.｣.....ﾝ.3ﾒｹ.@.....ｴ?ﾍ!.s.飫.｣.....ﾝ.ｺ.@ｹ.@.....ｴ?ﾍ!.s..6
0017C0:  .:...t.篤.=..t...ﾝ...ﾝ.ｾ..ｿ.@曲..ｦ.t・鯤.....ｴ>ﾍ!s.鵠.....ｴ>ﾍ!s.
001800:  鵐.鑚.ｺr.ｴ.ﾍ!ﾆ....<.u.ﾆ....縺.ｺ..ｴ.ﾍ!<.t..>...u.髓.鰥..>...t.驫.
001840:  .>...u..%声..ｴ;ﾍ!s.髯.ｺ..ｴ;ﾍ!s.髟.....罾.鴇.  ｾ)..>...t.ｾB..|...
001880:  ｾ...>...t.ｾ.........t....v.-#ｴ%ﾍ!..>x..t.ｺ..ｴ;ﾍ|.>...t..>y..t.ｺ&
0018C0:  .ｴ;ﾍ!.>...t.........ｴ+........ｴ-ﾍ!.....ｴ.ﾍ!ｴ.ﾍ!ｾ*.隰.  ｾM.陲.  .
001900:  2ﾒｴ2ﾍ!季..｣n....n.滙外.2  W....ﾛ.3ﾛﾍ%DD...ﾛ.ｻ.....n..ﾇ.p....?.u.
001940:  絵.$.<.u....p.|t.ε..?.u.絵.$.<pu....p.C|t.   .ﾃｺ..ｴ.ﾍ!ﾃｺ..   ..ｾ.
001980:  .ｿ7.ｹ...ｬ< t.ｪ欄ｾ..ｹ..ｬ< t.瑶-.ｪ監ｪ|ｬ< t.ｪ欄ﾆ..ﾃｾ"...ｾ....ｿﾛ.ｹ..
0019C0:  .ｬ< t.ｪ欄..|..ｹ..ｬ< t.瑶-.ｪ|監ｪｬ< t.ｪ欄ﾆ..ﾃ...ﾛ.ｾ.....ｾr......ﾛ.
001A00:  ｾ.....ｾ......ｹ(...､ﾃｹ,..､ｶ T.ﾃ...ｿ.....ｿr.......ｿ.....ｿ......ﾛ.
001A40:  T寓ｹ(...､ﾃｹ,..､.ﾃ.QRPｴ*ﾍ!........ｴ,ﾍ!.........諌...7.ﾓ閲.....
001A80:  .7.ﾓ閣ﾈ  ｼ.ｴ+ﾍ!...2ﾒ嬉...ﾐ竪.ﾓ関陟..7.ﾓ関ﾈ..?ｴ-ﾍ!ﾆ....XZY.ﾃ.....
001AC0:  ....ｴ+ﾍ!........ｴ-ﾍ!ﾆ.....ﾃ.....

A>
```

図 8-17　実行型ファイルの作成と実行結果

APPENDIX

◉ DEBUG（SYMDEB）の主要コマンド一覧 ◉

以下では，DEBUG (SYMDEB) の主要コマンドを本書で使用したものを中心にまとめています．コマンドの書式等については，通常よく使われる書式を取り上げ，それ以外の書式を示していないコマンドもありますから，注意してください．

なお，SYMDEB でのみ使えるコマンドについては，コマンド名の右肩に＊を付けて示しています．

DEBUG（SYMDEB）の起動

書式 1　**DEBUG（SYMDEB）**

DEBUG (SYMDEB) のみを単独で起動する．

書式 2　**DEBUG（SYMDEB）　<ファイル名>**

DEBUG (SYMDEB) の起動とともにプログラムをロードする．

書式の見方

[] で囲まれたパラメータは省略が可能である．

| でつながれたパラメータはいずれか一方を指定する．

<値>

・DEBUG の場合　……4 桁までの 16 進数

・SYMDEB の場合……2，8，10，16 進数で表された数，または式．

<アドレス>

セグメントレジスタまたは 4 桁までの 16 進数のセグメントアドレスによるセグメント指定と，4 桁までの 16 進数のオフセットアドレスによるオフセット指定を "：" で区切って指定する．

セグメント指定を省略した場合は，デフォルトのセグメントレジスタの内容が使われる．デフォルトのセグメントレジスタはコマンドによって異なり，Assemble, Go, Load, Unassemble, Write コマンドでは CS レジスタ，それ以外ではすべて DS レジスタとなる．

<レンジ>

メモリのアドレスの範囲を指定するもので，次の 2 通りがある．

① ＜アドレス１＞　＜アドレス２＞

② ＜アドレス＞　L＜値＞

①の指定方法では，＜アドレス1＞で開始アドレスを，＜アドレス2＞で終了アドレスを指定する．セグメント指定は＜アドレス1＞で指定されたものが＜アドレス2＞でも用いられ，＜アドレス2＞でのセグメント指定はできない．

②の指定方法は，Dump，Fill，Search，Unassemble コマンドでしか用いることができない．＜アドレス＞で開始アドレスを＜値＞で各コマンドの処理するバイト数などを指定する．＜値＞の最大値は 10000_H で 10000_H を指定する場合は 0 を用いる．SYMDEB はコマンドによって，指定する＜値＞がバイト数ではなく最大値もコマンドによって異なる場合がある．

＜リスト＞

任意個の＜値＞または引用符（「'」または「"」）で囲まれた任意長の文字列．

コマンド一覧

Assemble

書式　**A [＜アドレス＞]**

機能　8086／8087 の命令を 1 行ずつアセンブル*して，指定した＜アドレス＞以降のメモリに収める．

Compare

書式　**C＜レンジ＞　＜アドレス＞**

機能　＜レンジ＞で指定した範囲のメモリの内容と，＜アドレス＞からの同じ大きさの範囲のメモリの内容とを比較する．違いがあるときは，異なる部分のアドレスとその内容を表示する．

Dump

書式　**D [＜アドレス＞ | ＜レンジ＞]**

機能　指定されたアドレスから 128 バイト，または指定されたレンジのメモリの内容を，16 進値とアスキーキャラクタで表示する**.

---

* SYMDEB では，80186，80286／80287 の命令をアセンブルすることもできる．

** SYMDEB では，ASCII 型，バイト型，ワード型，倍長ワード型，4 バイト，8 バイト，10 バイトの各形式の浮動小数点型の 7 種類のダンプ形式（DA，DB，DW，……）があり，そのうち，直前に実行された型と同じ型で表示される．起動直後は，DEBUG と同じバイト型である．

Enter

書式　E＜アドレス＞　[＜リスト＞]

機能　メモリの指定されたアドレスにバイト値を入れる．
＜リスト＞を入力すると，＜アドレス＞から始まるメモリの内容が＜リスト＞の値に書き換えられる．また，＜アドレス＞のみを指定すると，そのアドレスを表示して入力待ちになるので，新しい値を 16 進数で入力する．

Fill

書式　F＜レンジ＞　＜リスト＞

機能　＜リスト＞で指定された値を繰り返し用いて，＜レンジ＞で指定された範囲のメモリを満たす．

Go

書式　G [=＜アドレス1＞]　[＜アドレス2＞……]

機能　メモリ上にロードされたプログラムを実行する．
＜アドレス1＞でスタートアドレスを，＜アドレス2＞以降でプログラムを中断または終了するアドレスを設定する．

Help*

書式　?

機能　SYMDEB のコマンドの一覧を表示する．

Hex

書式　H＜値1＞　＜値2＞

機能　2 つの 16 進数の和と差を計算する．

サンプル・オペレーション

```
-H AF 83↵ ……………0AFHと83Hの和と差を求める
Ø132  ØØ2C
-
        └── 差(0AFH－83H＝2CH)
 └──────── 和(0AFH＋83H＝0132H)
```

Input

書式　I＜値＞

機能　＜値＞で指定された I/O ポートの内容を表示する．

Load

書式　L [<アドレス>]

機能　ファイルの内容をメモリにロードする.

<アドレス>で指定されたアドレスから, また<アドレス>を省略した場合はCS：100Hから, Nコマンドで指定されたファイルをロードし, BX：CXに読み込んだバイト数をセットする.

サンプル・オペレーション

```
A>DEBUG↵ ……………DEBUGをパラメータなしで起動する                    ｝「DEBUG DISKCOPY.COM」
-N DISKCOPY.COM↵ ………ファイル名として「DISKCOPY.COM」をセットする  ｝として起動したのと
-L↵ ……………Nコマンドでセットしたファイル名のファイルをメモリ上にロードする ｝同じ結果になる
-S 0100 FFFF 'DISKCOPY'↵ ……………オフセットアドレス0100HからFFFFHまでのメモリから"DISKCOPY"というデータを探す
403A:012F ……………オフセットアドレス012FHで発見した
-D 012F 013F↵ ……………その付近をダンプする
403A:0120                                               44                 D
403A:0130  49 53 4B 43 4F 50 59 20-76 65 72 73 69 6F 6E 20  ISKCOPY version
-E 012F 'J'↵ ……………データを変更する
-D 012F 013F↵
403A:0120                                               4A                 J
403A:0130  49 53 4B 43 4F 50 59 20-76 65 72 73 69 6F 6E 20  ISKCOPY version
-W↵
Writing 1850 bytes
-Q↵
                                        3章を参考.
                                        結果的に同じことをやっている
A>DISKCOPY A: B:↵

JISKCOPY version 2.0

ディスクのコピーを行います

送り側ディスクをドライブ A: に挿入してください
受け側ディスクをドライブ B: に挿入してください
準備ができたらどれかのキーを押してください ^C

A>
```

Move

書式　M<レンジ>　<アドレス>

機能　<レンジ>で指定されたメモリ領域の内容を, 指定された<アドレス>から始まる領域へ転送する.

Name

書式　N<ファイル名>

機能　ファイル名をセットする.

サンプル・オペレーション　→Lコマンドを参照

Output

書式　O<値 1 >　<値 2 >

機能　<値 1 >で指定されたアドレスの I/O ポートに，<値 2 >を出力する．

Quit

書式　Q

機能　DEBUG（SYMDEB）を終了する．

Redirection*

書式　<　<ファイル名>｜<デバイス名>
　　　>　<ファイル名>｜<デバイス名>
　　　=　<デバイス名>

機能　SYMDEB のコマンド実行時の標準入出力を，指定されたファイルまたはデバイスに切り換える．

〝<〞と〝>〞は，それぞれ標準入力と標準出力を，〝=〞は標準入出力をともに指定されたファイルまたはデバイスに切り換える．

サンプル・オペレーション

```
A>TYPE ALPHA.A ↵
A 0100··············SYMDEBのコマンド
MOV AH,1                                    ALPHA.Aというファイル名で，エディタを使いこのような
INT 21                                      ファイルを作成しておく
CMP AL,41            6章実習10のプログラム
JB 0115                (図6-26参照)
CMP AL,5A
JA 0115
MOV AH,9
MOV DX,011E
INT 21
JMP 0110
MOV AH,9
MOV DX,013D
INT 21
JMP 0100
DB ' アルファベット大文字である',0D,0A,'$'
DB ' アルファベット大文字でない',0D,0A,'$'

<CON··············SYMDEBのコマンド

A>SYMDEB ↵
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [8086]
```

```
-<ALPHA.A↵ ……………以後の入力は「ALPHA.A」ファイルから行う
-A 0100…………………………………………………………………
3923:0100 MOV AH,1
3923:0102 INT 21
3923:0104 CMP AL,41
3923:0106 JB 0115
3923:0108 CMP AL,5A
3923:010A JA 0115
3923:010C MOV AH,9
3923:010E MOV DX,011E
3923:0111 INT 21
3923:0113 JMP 0110
3923:0115 MOV AH,9
3923:0117 MOV DX,013D
3923:011A INT 21
3923:011C JMP 0100
3923:011E DB '  アルファベット大文字である',0D,0A,'$'
3923:013D DB '  アルファベット大文字でない',0D,0A,'$'
3923:015C
-<CON…………………………………………………………………
-G↵ ──────→ 最後にこのコマンドがないと，コンソールに入力が戻らないので注意
C  アルファベット大文字である
3  アルファベット大文字でない
^C
AX=0124  BX=0000  CX=0000  DX=013D  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3923  ES=3923  SS=3923  CS=3923  IP=0104   NV UP EI NG NZ NA PO CY
3923:0104 3C41          CMP   AL,41                          ;'A'
-
```

「ALPHA.A」ファイルの内容によって自動実行される.
SYMDEBでプログラムを作成する際,「ALPHA.A」ファイルをエディタで編集することにより，プログラムの変更が容易になる

Register

書式 R [<レジスタ名> [[=]<値>]] *

機能 レジスタの内容を設定／表示する.

Search

書式 S<レンジ> <リスト>

機能 <レンジ>で指定された範囲で，指定した<リスト>を検索する.

サンプル・オペレーション →Lコマンドを参照

Shell Escape*

書式 ! [<MS-DOS コマンド>]

機能 SYMDEB のモードから，MS-DOS コマンドを実行する.<MS-DOS コマンド>を省略すると COMMAND.COM をロードし，制御を移す.

* DEBUG では,「R<レジスタ名>=<値>」によって，レジスタの値を設定することはできない．レジスタ名だけを指定することにより，新しい値の入力待ちになる.

サンプル・オペレーション

```
-!DIR A:↵ ……………SYMDEBの中から，MS-DOSのコマンドを実行する

 ドライブ A: のディスクのボリュームラベルはありません.
 ディレクトリは A:¥

COMMAND  COM    23942  85-12-11  15:28
CONFIG   SYS      105  86-11-08  19:57
SYS          <DIR>     86-11-18  22:46
BIN          <DIR>     86-11-18  22:46
AUTOEXEC BAT       31  86-11-08  19:43
         5 個のファイルがあります.
     51200 バイトが使用可能です.

-!↵ ……………パラメータなしで入力すると，COMMAND.COMが起動する

Command バージョン 3.10

A>DIR A:↵ ……………各種のコマンドが実行できる

 ドライブ A: のディスクのボリュームラベルはありません.
 ディレクトリは A:¥

COMMAND  COM    23942  85-12-11  15:28
CONFIG   SYS      105  86-11-08  19:57
SYS          <DIR>     86-11-18  22:46
BIN          <DIR>     86-11-18  22:46
AUTOEXEC BAT       31  86-11-08  19:43
         5 個のファイルがあります.
     51200 バイトが使用可能です.

A>EXIT↵ ……………「EXIT」コマンドで再びSYMDEBへ戻る
-        QコマンドでSYMDEBを終了した場合と異なり，メモリの内容などはすべて保存
         されているので，MS-DOSコマンドを実行する前の作業を続行することができる
```

Trace

書式 T [=<アドレス>]　[<値>]

機能 レジスタやフラグ，実行中の命令のニーモニックを表示しながら，1命令ずつ，<値>で指定された命令数だけプログラムを実行する.

Unassemble

書式 U [<レンジ>]

機能 <レンジ>で指定された範囲のメモリ内容を逆アセンブルして表示する.

Write

書式 W [<アドレス>]

機能 <アドレス>で指定されたアドレスから，また<アドレス>を省略した場合にはCS：100Hから，BX：CXで示されるバイト数だけ，メモリの内容をファイルとしてディスクに書き込む.

◉ キャラクタコード表 ◉

〈英数字，カナ，記号，漢字／全角文字の１バイト目〉

英大字と小文字はそれぞれ20Hずつ離れている（→202ページ）

31H　41H　35H　61H　81H　5AH　E0H

上位４ビット→　下位４ビット↓

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0		DE		0	@	P		p				ー	タ	ミ		
1	SH	D1	!	1	A	Q	a	q			。	ア	チ	ム		円
2	SX	D2	″	2	B	R	b	r			「	イ	ツ	メ		年
3	EX	D3	♯	3	C	S	c	s			」	ウ	テ	モ		月
4	ET	D4	$	4	D	T	d	t			、	エ	ト	ヤ		日
5	EQ	NK	%	5	E	U	e	u			・	オ	ナ	ユ		時
6	AK	SN	&	6	F	V	f	v			ヲ	カ	ニ	ヨ		分
7	BL	EB	'	7	G	W	g	w			ァ	キ	ヌ	ラ		秒
8	BS	CN	(	8	H	X	h	x			ィ	ク	ネ	リ	♠	
9	HT	EM	)	9	I	Y	i	y			ゥ	ケ	ノ	ル	♥	
A	LF	SB	*	:	J	Z	j	z			ェ	コ	ハ	レ	♦	
B	HM	EC	+	;	K	[	k	{			ォ	サ	ヒ	ロ	♣	
C	CL	→	,	<	L	¥	¦				ャ	シ	フ	ワ	●	
D	CR	←	−	=	M	]	m	}			ュ	ス	ヘ	ン	○	
E	SO	↑	.	>	N	∧	n	~			ョ	セ	ホ	゛		
F	SI	↓	/	?	O	_	o				ッ	ソ	マ	゜		

^Zのキャラクタコード（1AH）→115ページ

ベルを鳴らす（07H）

バックスペース（BS）のキャラクタコード（08H）→224ページ

タブキーのキャラクタコード（09H）

ラインフィード（0AH）CRと合わせて画面上での改行を行う

リターンキーのキャラクタコード（0DH）→160ページ

エスケープキーのキャラクタコード（1BH）

DDH　FCH　9FH

この部分の文字は，画面に表示できない文字である（→ 26ページ）．これらの文字を表示するというかたちで画面制御を行う

「JIS-C6220」で未定義のエリアであり，各社独自のグラフィック・キャラクタ等に使われている．シフトJIS漢字コードでは，このエリアのうち▭で囲んだ部分を漢字および全角文字の第１バイト目として使用している．

〈漢字／全角文字の２バイト目〉

上位４ビット→　下位４ビット↓

40H　FCH　7EH

▭で囲んだ部分がシフトJIS漢字コードの２バイト目として使われる部分

上表の▭の部分から１バイト目を下表の▭の部分から２バイト目を選ぶと，シフトJIS漢字コードとなる．そのうちいくつかを以下に示すが，くわしくはMS-DOSのマニュアル等にある漢字コード表を参照のこと．

上の表へ（１バイト目）

「A」 → 82 60H

「漢」 → 8A BFH

「字」 → 8E 9AH

◉ 10進-16進-2進 数値対応表 ◉

10進	16進	2進	10進	16進	2進	10進	16進	2進	10進	16進	2進
0	00	00000000	32	20	00100000	64	40	01000000	96	60	01100000
1	01	00000001	33	21	00100001	65	41	01000001	97	61	01100001
2	02	00000010	34	22	00100010	66	42	01000010	98	62	01100010
3	03	00000011	35	23	00100011	67	43	01000011	99	63	01100011
4	04	00000100	36	24	00100100	68	44	01000100	100	64	01100100
5	05	00000101	37	25	00100101	69	45	01000101	101	65	01100101
6	06	00000110	38	26	00100110	70	46	01000110	102	66	01100110
7	07	00000111	39	27	00100111	71	47	01000111	103	67	01100111
8	08	00001000	40	28	00101000	72	48	01001000	104	68	01101000
9	09	00001001	41	29	00101001	73	49	01001001	105	69	01101001
10	0A	00001010	42	2A	00101010	74	4A	01001010	106	6A	01101010
11	0B	00001011	43	2B	00101011	75	4B	01001011	107	6B	01101011
12	0C	00001100	44	2C	00101100	76	4C	01001100	108	6C	01101100
13	0D	00001101	45	2D	00101101	77	4D	01001101	109	6D	01101101
14	0E	00001110	46	2E	00101110	78	4E	01001110	110	6E	01101110
15	0F	00001111	47	2F	00101111	79	4F	01001111	111	6F	01101111
16	10	00010000	48	30	00110000	80	50	01010000	112	70	01110000
17	11	00010001	49	31	00110001	81	51	01010001	113	71	01110001
18	12	00010010	50	32	00110010	82	52	01010010	114	72	01110010
19	13	00010011	51	33	00110011	83	53	01010011	115	73	01110011
20	14	00010100	52	34	00110100	84	54	01010100	116	74	01110100
21	15	00010101	53	35	00110101	85	55	01010101	117	75	01110101
22	16	00010110	54	36	00110110	86	56	01010110	118	76	01110110
23	17	00010111	55	37	00110111	87	57	01010111	119	77	01110111
24	18	00011000	56	38	00111000	88	58	01011000	120	78	01111000
25	19	00011001	57	39	00111001	89	59	01011001	121	79	01111001
26	1A	00011010	58	3A	00111010	90	5A	01011010	122	7A	01111010
27	1B	00011011	59	3B	00111011	91	5B	01011011	123	7B	01111011
28	1C	00011100	60	3C	00111100	92	5C	01011100	124	7C	01111100
29	1D	00011101	61	3D	00111101	93	5D	01011101	125	7D	01111101
30	1E	00011110	62	3E	00111110	94	5E	01011110	126	7E	01111110
31	1F	00011111	63	3F	00111111	95	5F	01011111	127	7F	01111111

10進	16進	2進	10進	16進	2進	10進	16進	2進	10進	16進	2進
128	80	10000000	160	A0	10100000	192	C0	11000000	224	E0	11100000
129	81	10000001	161	A1	10100001	193	C1	11000001	225	E1	11100001
130	82	10000010	162	A2	10100010	194	C2	11000010	226	E2	11100010
131	83	10000011	163	A3	10100011	195	C3	11000011	227	E3	11100011
132	84	10000100	164	A4	10100100	196	C4	11000100	228	E4	11100100
133	85	10000101	165	A5	10100101	197	C5	11000101	229	E5	11100101
134	86	10000110	166	A6	10100110	198	C6	11000110	230	E6	11100110
135	87	10000111	167	A7	10100111	199	C7	11000111	231	E7	11100111
136	88	10001000	168	A8	10101000	200	C8	11001000	232	E8	11101000
137	89	10001001	169	A9	10101001	201	C9	11001001	233	E9	11101001
138	8A	10001010	170	AA	10101010	202	CA	11001010	234	EA	11101010
139	8B	10001011	171	AB	10101011	203	CB	11001011	235	EB	11101011
140	8C	10001100	172	AC	10101100	204	CC	11001100	236	EC	11101100
141	8D	10001101	173	AD	10101101	205	CD	11001101	237	ED	11101101
142	8E	10001110	174	AE	10101110	206	CE	11001110	238	EE	11101110
143	8F	10001111	175	AF	10101111	207	CF	11001111	239	EF	11101111
144	90	10010000	176	B0	10110000	208	D0	11010000	240	F0	11110000
145	91	10010001	177	B1	10110001	209	D1	11010001	241	F1	11110001
146	92	10010010	178	B2	10110010	210	D2	11010010	242	F2	11110010
147	93	10010011	179	B3	10110011	211	D3	11010011	243	F3	11110011
148	94	10010100	180	B4	10110100	212	D4	11010100	244	F4	11110100
149	95	10010101	181	B5	10110101	213	D5	11010101	245	F5	11110101
150	96	10010110	182	B6	10110110	214	D6	11010110	246	F6	11110110
151	97	10010111	183	B7	10110111	215	D7	11010111	247	F7	11110111
152	98	10011000	184	B8	10111000	216	D8	11011000	248	F8	11111000
153	99	10011001	185	B9	10111001	217	D9	11011001	249	F9	11111001
154	9A	10011010	186	BA	10111010	218	DA	11011010	250	FA	11111010
155	9B	10011011	187	BB	10111011	219	DB	11011011	251	FB	11111011
156	9C	10011100	188	BC	10111100	220	DC	11011100	252	FC	11111100
157	9D	10011101	189	BD	10111101	221	DD	11011101	253	FD	11111101
158	9E	10011110	190	BE	10111110	222	DE	11011110	254	FE	11111110
159	9F	10011111	191	BF	10111111	223	DF	11011111	255	FF	11111111

◉ 1バイトおよび1ワードデータの16進-10進変換法 ◉

BASICで以下のように実行すればよい

10進 → 16進変換

```
Ok              10進数
PRINT HEX$(8Ø)
5Ø ←  16進数
Ok
PRINT HEX$(128)
8Ø
Ok
PRINT HEX$(256)
1ØØ
Ok
PRINT HEX$(1Ø24)
4ØØ
Ok
```

プログラムを組む必要はなく，ダイレクトに実行すればよい

16進 → 10進変換

```
1Ø INPUT A$
2Ø A=VAL("&H"+A$)
3Ø IF A<Ø THEN A=A+A^16
4Ø PRINT A
5Ø GOTO 1Ø
RUN
? 4A       16進数
 74        10進数
? 8Ø
 128
? FF
 255
? 1234
 466Ø
?
```

符号付きの数を扱いたいときはこの行を削除

16進数 → 10進数変換プログラム

◉ 8086主要インストラクションセット ◉

〈表の見方〉

8086CPUの主要なインストラクションセット(命令の組合せ)を，本文での解説と同じように機能別に分類し，表にしてあります．各命令のニーモニックは，「オペランド」*の部分を一般的な形で示しています．実際の使用にあたっては，次に示すオペランドの例を参照してください．

また，各命令の使用例はすべてMASMで使用する書式としてまとめています．このうちラベルを使って表したものについては，DEBUGでは直接アドレス値を指定しなければならないことに注意してください．

*オペランドは各命令に必要なパラメータのこと(92ページ参照)．

オペランド

オペランドの略号		実際の指定例
即　値		1234H，OFEH OFFSET ASCII　など
メモリ		データラベル*(COUNT，MESSAGE　など) [BX]，[BP]，[BX+9876H]，[BP+3]，[SI]，[DI]，[SI+0FH]，[DI+0FEH]，[BP+SI]，[BP+DI]，[BX+SI]，[BX+DI]，[BP+SI+1234H]，[BP+DI+32H]，[BX+SI+0FFH]，[BX+DI+0010H]　など
セグメントレジスタ		CS，DS，ES，SS
汎用レジスタ	16ビット	AX，BX，CX，DX，BP，SP，SI，DI
	8ビット	AH，AL，BH，BL，CH，CL，DH，DL
コードラベル*		ASCII，NOT_ASCII， JIS，NOT_JIS， ALPHA，BETA　など

*データラベル，コードラベルについては本文244ページを参照．

1．データ転送命令

ニーモニック			機　能	使用例	本文参照ページ
命令	第1オペランド	第2オペランド			
MOV	汎用レジスタ	，汎用レジスタ	第1オペランド ← 第2オペランド（CSを除く） 第2オペランドの内容を第1オペランドへ転送する．	MOV AX，CX	
MOV	メモリ	，汎用レジスタ		MOV [DI]，BX	129
MOV	汎用レジスタ	，メモリ		MOV CX，COUNT	131
MOV	メモリ	，即値			138
MOV	汎用レジスタ	，即値		MOV DX，0200H	128
MOV	セグメントレジスタ	，16ビット汎用レジスタ		MOV DS，AX	134
MOV	セグメントレジスタ	，メモリ		MOV ES，VRAM_SEG	
MOV	16ビット汎用レジスタ	，セグメントレジスタ		MOV AX，CS	
MOV	メモリ	，セグメントレジスタ		MOV SEG_BASE，DS	
XCHG	汎用レジスタ	汎用レジスタ	第1オペランド ↔ 第2オペランド 第1オペランドと第2オペランドの値を入れ換える．	XCHG BX，CX	
XCHG	メモリ	汎用レジスタ		XCHG POSITION_X，DX	

〈フラグの変化〉
データ転送命令では，フラグは変化しない．

2．算術演算命令

ニーモニック			機　能	使用例	本文参照ページ
命令	第1オペランド	第2オペランド			
ADC	汎用レジスタ	，汎用レジスタ	第1オペランド ← 第1オペランド ＋ 第2オペランド ＋ CF 第1オペランドにキャリーフラグと第2オペランドの値を加算する．	ADC AX，DX	
ADC	メモリ	，汎用レジスタ		ADC [BX]，AX	
ADC	汎用レジスタ	，メモリ		ADC DX，[SI]	
ADC	汎用レジスタ	，即値		ADC BX，0256H	
ADC	メモリ	，即値			
ADD	汎用レジスタ	，汎用レジスタ	第1オペランド ← 第1オペランド ＋ 第2オペランド 第1オペランドに第2オペランドの値を加算する	ADD CX，DX	
ADD	メモリ	，汎用レジスタ		ADD [DI]，BX	
ADD	汎用レジスタ	，メモリ		ADD AH，[SI]	
ADD	汎用レジスタ	，即値		ADD BX，0256H	147
ADD	メモリ	，即値			
CMP	汎用レジスタ	，汎用レジスタ	← 第1オペランド － 第2オペランド 第1オペランドと第2オペランドの値を比較する．	CMP DL，AL	
CMP	メモリ	，汎用レジスタ		CMP [BX]，DL	
CMP	汎用レジスタ	，メモリ		CMP DX，[SI＋6]	
CMP	汎用レジスタ	，即値		CMP AL，0DH	151
CMP	メモリ	，即値			
DEC	汎用レジスタ		汎用レジスタの値を1減じる．	DEC AX	149
DEC	メモリ		指定されたメモリの値を1減じる．		
DIV	汎用レジスタ		8ビット　商 AL，余り AH ← AX ÷ 第1オペランド	DIV CL	153
DIV	メモリ		16ビット　商 AX，余り DX ← DX AX ÷ 第1オペランド	DIV BX	
IDIV	汎用レジスタ		符号付きで除算を行うこと以外は，基本的にはDIVと同じ．	IDIV BX	
IDIV	メモリ				
IMUL	汎用レジスタ		符号付きで乗算を行うこと以外は，基本的にはMULと同じ．	IMUL DX	
IMUL	メモリ				

ニーモニック			機　　能	使用例	本文参照ページ
命令	第1オペランド	第2オペランド			
INC INC	汎用レジスタ メモリ		汎用レジスタの値を1増やす. 指定されたメモリの値を1増やす.	INC　AX	149
MUL MUL	汎用レジスタ メモリ		8ビット AX ← AL × 第1オペランド 16ビット DX AX ← AX × 第1オペランド	MUL　CL MUL　BX	153
SBB SBB SBB SBB SBB	汎用レジスタ メモリ 汎用レジスタ 汎用レジスタ メモリ	汎用レジスタ 汎用レジスタ メモリ 即値 即値	第1オペランド ← 第1オペランド － 第2オペランド － CF 第1オペランドからキャリーフラグと第2オペランドの値を減ずる.	SBB　AX,CX SBB　[BP+6],CX SBB　DX,[SI−2] SBB　BX,0256H	
SUB SUB SUB SUB SUB	汎用レジスタ メモリ 汎用レジスタ 汎用レジスタ メモリ	汎用レジスタ 汎用レジスタ メモリ 即値 即値	第1オペランド ← 第1オペランド － 第2オペランド 第1オペランドから第2オペランドの値を減ずる.	SUB　SI,CX SUB　[BX+4],CL SUB　BX,256	147

〈フラグの変化〉

命令	O	D	I	T	S	Z	A	P	C
ADC	X				X	X	X	X	X
ADD	X				X	X	X	X	X
CMP	X				X	X	X	X	X
DEC	X				X	X	X	X	
INC	X				X	X	X	X	
DIV	U				U	U	U	U	U
IDIV	U				U	U	U	U	U
MUL	X				U	U	U	U	X
IMUL	X				U	U	U	U	X
SBB	X				X	X	X	X	X
SUB	X				X	X	X	X	X

フラグ
- O：オーバーフローフラグ
- D：ディレクションフラグ
- I：インタラプトイネーブルフラグ
- T：トラップフラグ
- S：サインフラグ
- Z：ゼロフラグ
- A：補助キャリーフラグ
- P：パリティフラグ
- C：キャリーフラグ

フラグの値
- 0：フラグが0にリセットされる.
- 1：フラグが1にセットされる.
- X：結果によって0にリセットまたは1にセットされる.
- U：どうなるか分からない.
- R：フラグを復帰する.
- 空白：変化しない.

3.ジャンプ命令

ニーモニック		機　　能	使用例	本文参照ページ
命令	オペランド			
JA JNBE	コードラベル コードラベル	キャリーフラグ,ゼロフラグがともに0のとき,コードラベルで示されるアドレスへジャンプする.	JA　ASCII JNBE　JIS	169
JAE JNB	コードラベル コードラベル	キャリーフラグが0のとき，コードラベルで示されるアドレスへジャンプする.	JAE　JAPAN JNB　USA	169
JB JNAE	コードラベル コードラベル	キャリーフラグが1のとき，コードラベルで示されるアドレスへジャンプする.	JB　NOT_USA JNAE　NOT_JAPAN	169
JBE JNA	コードラベル コードラベル	キャリーフラグ,またはゼロフラグが1のとき,コードラベルで示されるアドレスへジャンプする.	JBE　NOT_JIS JNA　NOT_ASCII	169

ニーモニック		機　　　能	使用例	本文参照ページ
命令	オペランド			
JC	コードラベル	キャリーフラグが1のとき，コードラベルで示されるアドレスへジャンプする.	JC OPEN	
JCXZ	コードラベル	CXレジスタが0のとき，コードラベルで示されるアドレスへジャンプする.	JCXZ SYSTEM	
JE JZ	コードラベル コードラベル	ゼロフラグが1のとき，コードラベルで示されるアドレスへジャンプする.	JE OPENING JZ ENDING	169 167
JG JNLE	コードラベル コードラベル	サインフラグ，オーバーフローフラグがともに0または1で,かつゼロフラグが0のとき,コードラベルで示されるアドレスへジャンプする.	JG PROLOG JNLE LISP	169
JGE JNL	コードラベル コードラベル	サインフラグ，オーバーフローフラグがともに0または1のとき，コードラベルで示されるアドレスへジャンプする.	JGE JUMP JNL STEP	169
JL JNGE	コードラベル コードラベル	サインフラグ，オーバーフローフラグのどちらか一方が0で他方が1のとき，コードラベルで示されるアドレスへジャンプする.	JL YOUNG JNGE OLD	169
JLE JNG	コードラベル コードラベル	サインフラグ，オーバーフローフラグのどちらか一方が0で他方が1，またはゼロフラグが1のとき，コードラベルで示されるアドレスへジャンプする.	JLE WORLD_W JNG WORLD_L	169
JMP	コードラベル	コードラベルで示されるアドレスへジャンプする.	JMP EXIT	162
JNC	コードラベル	キャリーフラグが0のとき，コードラベルで示されるアドレスへジャンプする	JNC STEP_1	
JNE JNZ	コードラベル コードラベル	ゼロフラグが0のとき，コードラベルで示されるアドレスへジャンプする.	JNE UTILTY JNZ SOURCE	169
LOOP	コードラベル	CXレジスタの値を1減じて，CXレジスタが0でなければコードラベルで示されるアドレスへジャンプする.	LOOP TOP	176
LOOPE LOOPZ	コードラベル コードラベル	CXレジスタの値を1減じて，CXレジスタが0でなく，ゼロフラグが1のとき，コードラベルで示されるアドレスへジャンプする.	LOOPE LOOP_TOP LOOPZ LOOP_START	
LOOPNE LOOPNZ	コードラベル コードラベル	CXレジスタの値を1減じて，CXレジスタが0でなく，ゼロフラグが0のとき，コードラベルで示されるアドレスへジャンプする.	LOOPNE AGAIN LOOPNZ MORE	

〈フラグの変化〉
ジャンプ命令では，フラグは変化しない.

4.コール・リターン／スタック操作命令

ニーモニック		機　　　能	使用例	本文参照ページ
命令	オペランド			
CALL	コードラベル	コードラベルで示されるサブルーチンをコールする.	CALL DISPLAY	183

ニーモニック		機　　能	使用例	本文参照ページ
命令	オペランド			
RET RET	 スタックから捨てるバイト数	CALL命令でコールされたサブルーチンから復帰する． スタックから指定されたバイト数分のデータを捨ててから復帰する．	RET RET　10	183

POP POP POP	16ビット汎用レジスタ セグメントレジスタ(CSを除く) メモリ	スタックから指定されたオペランドへ値を復帰する．	POP　CX POP　DS POP　[BP+DI]	186
POPF		フラグレジスタをスタックから復帰する．	POPF	187
PUSH PUSH PUSH	16ビット汎用レジスタ セグメントレジスタ メモリ	指定されたオペランドの値をスタックへ退避する．	PUSH　BP PUSH　ES PUSH　[BP+DI]	186
PUSHF		フラグレジスタをスタックへ退避する．	PUSHF	187

〈フラグの変化〉
　コール・リターン／スタック操作命令では，フラグは変化しない．

5．ストリング／リピートプリフィックス命令

ニーモニック	機　　能	使用例	本文参照ページ
CMPSB	DS：[SI]とES：[DI]で示されるメモリをバイト値で比較し，SIとDIの値を1増加または減少させる．	CMPSB	197
CMPSW	DS：[SI]とES：[DI]で示されるメモリをワード値で比較し，SIとDIの値を2増加または減少させる．	CMPSW	197
LODSB	DS：[SI]で示されるメモリのバイト値をALレジスタへ転送し，SIの値を1増加または減少させる．	LODSB	197
LODSW	DS：[SI]で示されるメモリのワード値をAXレジスタへ転送し，SIの値を2増加または減少させる．	LODSW	197
MOVSB	DS：[SI]で示されるメモリのバイト値をES：[DI]で示されるメモリへ転送し，SI，DIの値を1増加または減少させる．	MOVSB	192 197
MOVSW	DS：[SI]で示されるメモリのワード値をES：[DI]で示されるメモリへ転送し，SI，DIの値を2増加または減少させる．	MOVSW	195 197
SCASB	ES：[DI]で示されるメモリのバイト値とALレジスタの値を比較し，DIの値を1増加または減少させる．	SCASB	197
SCASW	ES：[DI]で示されるメモリのワード値とAXレジスタの値を比較し，DIの値を2増加または減少させる．	SCASW	197

REP	CXレジスタの値が0になるまで，次に続くストリング命令を繰り返す．CXレジスタの内容を繰り返しのたびに1減じる．	REP　MOVSB	192 197
REPE REPZ	ゼロフラグが1で，かつCXレジスタが0でない間，次に続くストリング命令を繰り返す．CXレジスタの内容を繰り返しのたびに1減じる．	REPE　CMPSB REPZ　CMPSW	197
REPNE REPNZ	ゼロフラグが0で，かつCXレジスタが0でない間，次に続くストリング命令を繰り返す．CXレジスタの内容を繰り返しのたびに1減じる．	REPNE　SCASB REPNZ　SCASW	197

ニーモニック	機　　能	使用例	本文参照ページ
CLD	ディレクションフラグを0(増加)にリセットする.	CLD	194
STD	ディレクションフラグを1(減少)にセットする.	STD	195

〈注意〉
・DS:[SI]はDSレジスタの内容をセグメントアドレスとし, SIレジスタの内容をオフセットアドレスとするメモリの内容という意味. 同様にES:[DI]はESレジスタ, DIレジスタの内容をそれぞれセグメントアドレス, オフセットアドレスとするメモリの内容を表す.
・増減の方向はディレクションフラグによって決まる.

〈フラグの変化〉
以下の命令のみ, フラグが変化する.

命令	フラグ O	D	I	T	S	Z	A	P	C	命令	フラグ O	D	I	T	S	Z	A	P	C
CLD		0								STD		1							

6.論理演算命令

ニーモニック 命令	第1オペランド	第2オペランド	機　　能	使用例	本文参照ページ
AND AND AND AND AND	汎用レジスタ メモリ 汎用レジスタ 汎用レジスタ メモリ	汎用レジスタ 汎用レジスタ メモリ 即値 即値	第1オペランドと第2オペランドの各ビットごとにANDをとり結果を第1オペランドに格納する.	AND BX,CX AND BITS,AX AND DL,[DI] AND AL,05FH AND WORD PTR [BP+3],0FH	200
NEG NEG	汎用レジスタ メモリ		オペランドの値の符号を反転する.	NEG AX NEG BITS	
NOT NOT	汎用レジスタ メモリ		オペランドの値のすべてのビットを反転させる.	NOT DX NOT WORD PTR [BP+SI]	
OR OR OR OR OR	汎用レジスタ メモリ 汎用レジスタ 汎用レジスタ メモリ	汎用レジスタ 汎用レジスタ メモリ 即値 即値	第1オペランドと第2オペランドの各ビットごとにORをとり結果を第1オペランドに格納する.	OR AL,AH OR MSB,AX OR DX,[BP−6] OR AL,20H OR WORD PTR [SI],8001H	200
TEST TEST TEST TEST	汎用レジスタ メモリ 汎用レジスタ メモリ	汎用レジスタ 汎用レジスタ 即値 即値	ANDと同じ演算を行い, フラグを変化させるが, もとのレジスタの値は変化させない.	TEST AX,CX TEST RSDATA,AX TEST DL,02H TEST BYTE PTR [SI+3],80H	
XOR XOR XOR XOR XOR	汎用レジスタ メモリ 汎用レジスタ 汎用レジスタ メモリ	汎用レジスタ 汎用レジスタ メモリ 即値 即値	第1オペランドと第2オペランドの各ビットごとにXORをとり結果を第1オペランドに格納する.	XOR AX,AX XOR LSB,DX XOR DL,[DI] XOR CH,0FH XOR WORD PTR [DI],0004H	

〈フラグの変化〉

命令	フラグ O	D	I	T	S	Z	A	P	C	命令	フラグ O	D	I	T	S	Z	A	P	C
AND	0				X	X	U	X	0	OR	0				X	X	U	X	0
NEG	X				X	X	X	X	1	TEST	0				X	X	U	X	0
NOT										XOR	0				X	X	U	X	0

7.シフト・ローテイト命令

ニーモニック 命令	第1オペランド	第2オペランド	機能	使用例	本文参照ページ
RCL RCL	汎用レジスタ メモリ	,1またはCL ,1またはCL	第1オペランド CF ← MSB ← LSB ← キャリーを含めたビットの左回転. MSB=最上位ビット LSB=最下位ビット	RCL AX,1 RCL BITS,CL	205
RCR RCR	汎用レジスタ メモリ	,1またはCL ,1またはCL	第1オペランド → MSB → LSB → CF キャリーを含めたビットの右回転.	RCR BX,CL RCR WORD PTR [SI],1	205
ROL ROL	汎用レジスタ メモリ	,1またはCL ,1またはCL	第1オペランド CF ← MSB ← LSB ← ビットの左回転.	ROL AX,1 ROL ALL,CL	205
ROR ROR	汎用レジスタ メモリ	,1またはCL ,1またはCL	第1オペランド → MSB → LSB → CF ビットの右回転.	ROR BX,CL ROR BYTE PTR [BX+2],1	205 206
SAL SAL SHL SHL	汎用レジスタ メモリ 汎用レジスタ メモリ	,1またはCL ,1またはCL ,1またはCL ,1またはCL	第1オペランド CF ← MSB ← LSB ← 0 ビットの左シフト. 最下位ビットには0が入る.	SAL AL,1 SAL LEFT,CL SHL BX,CL SHL PATTERN,1	205 205
SAR SAR	汎用レジスタ メモリ	,1またはCL ,1またはCL	第1オペランド → MSB → LSB → CF ビットの右シフト. 最上位ビットはそのまま.	SAR AX,1 SAR BTO,CL	205
SHR SHR	汎用レジスタ メモリ	,1またはCL ,1またはCL	第1オペランド 0 → MSB → LSB → CF ビットの右シフト. 最上位ビットには0が入る.	SHR BX,1 SHR WORD PTR [SI+3],CL	205

〈フラグの変化〉

命令	O	D	I	T	S	Z	A	P	C	命令	O	D	I	T	S	Z	A	P	C
RCL	X								X	SAL	X								X
RCR	X								X	SAR	X				X	X	U	X	X
ROL	X								X	SHL	X								X
ROR	X								X	SHR	X								X

8.入出力命令

ニーモニック			機能	使用例	本文参照ページ
命令	第1オペランド	第2オペランド			
IN IN	AXまたはAL AXまたはAL	ポート番号 DX	指定されたI/Oポートからデータを入力し，AXまたはALレジスタに格納する.	IN AX,37H IN AL,DX	208
OUT OUT	ポート番号 DX	AXまたはAL AXまたはAL	指定されたI/OポートへAXまたはALレジスタの値を出力する.	OUT 37H,AX OUT DX,AL	209

〈フラグの変化〉
入出力命令では，フラグは変化しない.

9.割り込み命令

ニーモニック	機能	使用例	本文参照ページ
CLI	インタラプトイネーブルフラグを0にリセットする. (外部割り込みを禁止状態にする.)	CLI	
INT 番号	指定された番号のソフトウェア割り込みを発生する.	INT 21H	216
IRET	割り込み処理ルーチンから復帰する.	IRET	
STI	インタラプトイネーブルフラグを1にセットする. (外部割り込みを受け付け可能にする.)	STI	

〈フラグの変化〉

命令	フラグ O	D	I	T	S	Z	A	P	C	命令	フラグ O	D	I	T	S	Z	A	P	C
CLI			0							IRET	R	R	R	R	R	R	R	R	R
INT			0	0						STI			1						

10.その他の命令

ニーモニック	機能	使用例	本文参照ページ
HLT	割り込みがかかるまでCPUを停止させる.	HLT	
NOP	何もしない.	NOP	

〈フラグの変化〉
その他の命令では，フラグは変化しない.

索　　引

8086CPU のマシン語命令の索引は，276 ページからの「8086 主要インストラクションセット」を参照のこと．

カ

サ

タ

ナ

ハ

マ

ラ

ワ

参考文献

・「iAPX86　マクロアセンブリ言語プログラミングマニュアル」，
インテルジャパン株式会社　CQ 出版

・「MS-DOS ユーザーズマニュアル」　NEC

・「MS-DOS マクロアセンブラマニュアル」　NEC

・「はじめて読むマシン語」　村瀬康治著　アスキー出版局

・「応用 MS-DOS」　村瀬康治著　アスキー出版局

・「標準 MS-DOS ハンドブック」　アスキー出版局編著　アスキー出版局

・「80286 ハンドブック」　大貫，田中，蓑原共著　アスキー出版局

・「PC-9801 マシン語入門」　岩瀬，藤井共著　アスキー出版局

・「PC-9800 シリーズ　テクニカルデータブック」　アスキー出版局テクライト編
アスキー出版局

・「16 ビットプログラミング技法＜8086 プログラムの組み方＞」　村山仁郎著
産報出版

・「V30 プログラマーズリファレンス」　ビー・エヌ・エヌ企画部　BNN

・「32 ビット・マイクロプロセッサ入門」　南宗宏　CQ 出版

・「MS-DOS マクロアセンブラ入門」　藤木文彦著　ナツメ社

商　標

・MS-DOSは，米MICROSOFT社の登録商標です．

・V30は，日本電気株式会社の商標です．

・Z80は，米Zilog, Inc.の商標です．

その他，プログラム名，システム名，CPU名は一般に各開発メーカーの登録商標です．

なお，本文中ではTM，Ⓡマークは明記していません．

はじめて読む8086

1987年4月1日　初版発行
2001年5月21日　第1版第35刷発行

著　者　蒲池（かまち）輝尚（てるひさ）
監　修　村瀬（むらせ）康治（やすはる）
発行人　鈴木 憲一
編集人　土屋 信明
発行所　**株式会社アスキー**
〒151-8024　東京都渋谷区代々木4-33-10
大代表　(03)5351-8111
出版営業部　(03)5351-8194（ダイヤルイン）
APC書籍編集部　(03)5351-8106（ダイヤルイン）

制　作　株式会社 GARO
印　刷　株式会社 加藤文明社印刷所

編　集　佐藤 英一

ISBN4-87148-245-6　　**Printed in Japan**

●11172／11411

はじめて読むシリーズ

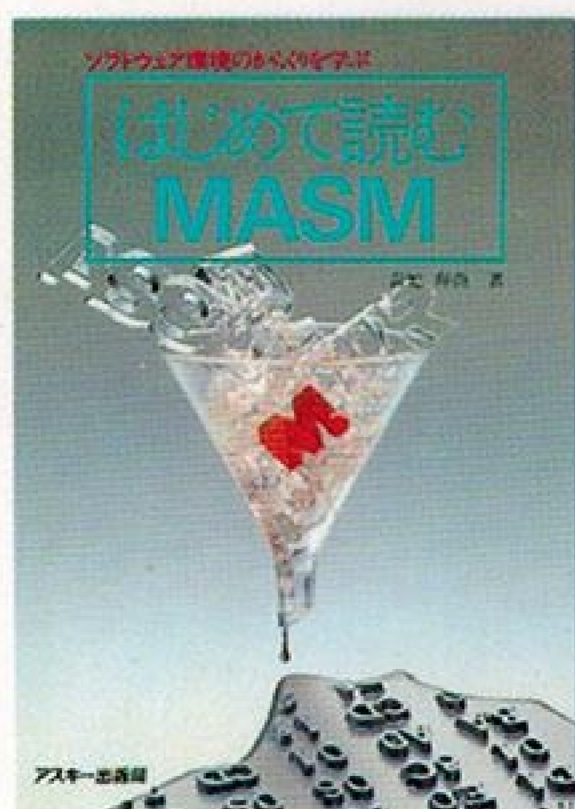
はじめて読むC言語



蒲地輝尚 著　定価 本体1,748円 +税

標準的なプログラミング環境であるC言語を，その基礎から徹底解説．やさしい図説を読み進めるうちに，C言語は難しいと気後れしていたユーザーでも，自然にプログラムを組む力が身に付きます．全Cユーザー必読の1冊．

はじめて読むMASM



蒲地輝尚 著　定価 本体1,796円 +税

本書は「はじめて読む8086」の続編です．難解なMASMの擬似命令やセグメントの概念を，嚙み砕いた解説と豊富な図版で語ります．本格的なプログラム事例も掲載しているので，これからMASMを学ぼうとする読者に最適の書籍です．

はじめて読むアセンブラ



村瀬康治 著　定価 本体1,602円 +税

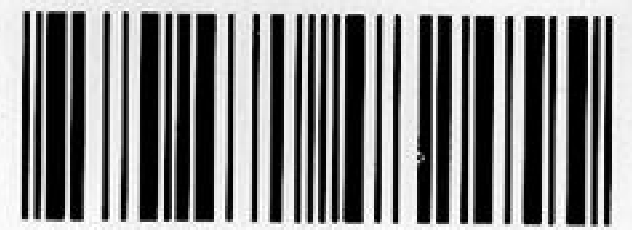
本書は「はじめて読むマシン語」の続編です．本格的なプログラム作りを目指す人のために，アセンブラの全体像をやさしく解説．CP/M上の各種ツールの使い方を学びながら，ソフトウェア開発の基礎をしっかりと把握できます．

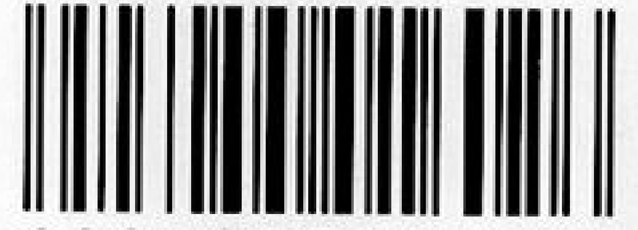
はじめて読むマシン語



村瀬康治 著　定価 本体1,204円 +税

はじめてマシン語を学ぶ人のための啓蒙的入門書．コンピュータの基礎知識もあわせて解説していますから，初心者でも十分に読みこなすことができます．PC-8801シリーズ，X1シリーズ，MSXなどZ80，8080系マシンユーザー必携．





ISBN4-87148-245-6

C3055 ¥1602E

定価 本体1,602円 +税

